BlueXP automation catalog

NetApp Automation

NetApp
February 11, 2024

Table of Contents

BlueXP automation catalog

Overview of the BlueXP automation catalog

The BlueXP automation catalog is a collection of automation solutions available to NetApp customers, partners, and employees. Some of the features and benefits of the catalog are described below.

Single location for your automation needs

You can access the BlueXP automation catalog through the BlueXP web user interface. This provides a single location for the scripts, playbooks, and modules needed to enhance the automation and operation of your NetApp products and services.

Solutions are created and tested by NetApp

All the automation solutions and scripts have been created and tested by NetApp. Each solution targets a specific customer use case or request. Most focus on integration with the NetApp file and data services.

Documentation

Each of the automation solutions includes associated documentation to help you get started. While the solutions are accessed through the BlueXP web interface, all the documentation is available at this site. The documentation is organized based on the NetApp products and cloud services.

Solid foundation for the future

NetApp is committed to helping our customers improve and streamline the automation of their data centers and cloud environments. We expect to continue enhancing the BlueXP automation catalog to address customer requirements, technology changes, and continued product integration.

We want to hear from you

The NetApp Customer Experience Office (CXO) automation team would like to hear from you. If you have any feedback, issues, or feature requests, please send an email to CXO automation team.

Amazon FSx for NetApp ONTAP

Amazon FSx for NetApp ONTAP - Burst to cloud

You can use this automation solution to provision Amazon FSx for NetApp ONTAP with volumes and an associated FlexCache.

Amazon FSx for NetApp ONTAP is also referred to as **FSx for ONTAP**.

About this solution

At a high level, the automation code provided with this solution performs the following actions:

- Provision a destination FSx for ONTAP file system
- Provision Storage Virtual Machines (SVMs) for the file system
- Create a cluster peering relationship between the source and destination systems
- Create an SVM peering relationship between the source system and destination system for FlexCache

- Optionally create FlexVol volumes using FSx for ONTAP
- Create a FlexCache volume in FSx for ONTAP with the source pointing to on-prem storage

The automation is based on Docker and Docker Compose which must be installed on the Linux virtual machine as described below.

Before you begin

You must have the following to complete the provisioning and configuration:

- You need to download the Amazon FSx for NetApp ONTAP - Burst to cloud automation solution through the BlueXP web UI. The solution is packaged as file `AWS_FSxN_BTC.zip`.
- Network connectivity between the source and destination systems.
- A Linux VM with the following characteristics:
  - Debian-based Linux distribution
  - Deployed on the same VPC subset used for FSx for ONTAP provisioning
- AWS account.

Step 1: Install and configure Docker

Install and configure Docker in a Debian-based Linux virtual machine.

Steps

1. Prepare the environment.

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gnupg-
agent software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt-get update
```

2. Install Docker and verify the installation.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
docker --version
```

3. Add the required Linux group with an associated user.

   First check if the group **docker** exists in your Linux system. If it doesn't, create the group and add the user. By default, the current shell user is added to the group.

```
sudo groupadd docker
sudo usermod -aG docker $(whoami)
```

4. Activate the new group and user definitions

   If you created a new group with a user, you need to activate the definitions. To do this, you can logout of Linux and then back in. Or you can run the following command.

```
newgrp docker
```

Step 2: Install Docker Compose

Install Docker Compose in a Debian-based Linux virtual machine.

Steps

1. Install Docker Compose.

```
sudo curl -L
"https://github.com/docker/compose/releases/latest/download/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

2. Verify the installation was successful.

```
docker-compose --version
```

Step 3: Prepare the Docker image

You need to extract and load the Docker image provided with the automation solution.

Steps

1. Copy the solution file `AWS_FSxN_BTC.zip` to the virtual machine where the automation code will run.

```
scp -i ~/<private-key.pem> -r AWS_FSxN_BTC.zip user@<IP_ADDRESS_OF_VM>
```

   The input parameter `private-key.pem` is your private key file used for AWS virtual machine authentication (EC2 instance).

2. Navigate to the correct folder with the solution file and unzip the file.

```
unzip AWS_FSxN_BTC.zip
```

3. Navigate to the new folder `AWS_FSxN_BTC` created with the unzip operation and list the files. You should see file `aws_fsxn_flexcache_image_latest.tar.gz`.

```
ls -la
```

4. Load the Docker image file. The load operation should normally complete in a few seconds.

```
docker load -i aws_fsxn_flexcache_image_latest.tar.gz
```

5. Confirm the Docker image is loaded.

```
docker images
```

You should see the Docker image `aws_fsxn_flexcache_image` with the tag `latest`.

```
   REPOSITORY              TAG     IMAGE ID      CREATED      SIZE
aws_fsxn_flexcahce_image  latest  ay98y7853769  2 weeks ago  1.19GB
```

Step 4: Create environment file for AWS credentials

You must create a local variable file for authentication using the access and secret key. Then add the file to the `.env` file.

Steps

1. Create the `awsauth.env` file in the following location:

   `path/to/env-file/awsauth.env`

2. Add the following content to the file:

```
access_key=<>
secret_key=<>
```

   The format **must** be exactly as shown above without any spaces between `key` and `value`.

3. Add the absolute file path to the `.env` file using the `AWS_CREDS` variable. For example:

   `AWS_CREDS=path/to/env-file/awsauth.env`

Step 5: Create an external volume

You need an external volume to make sure the Terraform state files and other important files are persistent. These files must be available for Terraform to run the workflow and deployments.

Steps

1. Create an external volume outside of Docker Compose.

   Make sure to update the volume name (last parameter) to the appropriate value before running the command.

   ```
   docker volume create aws_fsxn_volume
   ```

2. Add the path to the external volume to the `.env` environment file using the command:

   `PERSISTENT_VOL=path/to/external/volume:/volume_name`

   Remember to keep the existing file contents and colon formatting. For example:

   ```
   PERSISTENT_VOL=aws_fsxn_volume:/aws_fsxn_flexcache
   ```

   You can instead add an NFS share as the external volume using a command such as:

   `PERSISTENT_VOL=nfs/mnt/document:/aws_fsx_flexcache`

3. Update the Terraform variables.
   a. Navigate to the folder `aws_fsxn_variables`.
   b. Confirm the following two files exist: `terraform.tfvars` and `variables.tf`.
   c. Update the values in `terraform.tfvars` as required for your environment.

      See Terraform resource: aws_fsx_ontap_file_system for more information.

Step 6: Provision Amazon FSx for NetApp ONTAP and FlexCache

You can provision Amazon FSx for NetApp ONTAP and FlexCache.

Steps

1. Navigate to the folder root (AWS_FSXN_BTC) and issue the provisioning command.

   ```
   docker-compose -f docker-compose-provision.yml up
   ```

   This command creates two containers. The first container deploys FSx for ONTAP and the second container creates the cluster peering, SVM peering, destination volume, and FlexCache.

2. Monitor the provisioning process.

   ```
   docker-compose -f docker-compose-provision.yml logs -f
   ```

   This command gives you the output in real time, but has been configured to capture the logs through the file `deployment.log`. You can change the name of these log files by editing the `.env` file and updating

the variables `DEPLOYMENT_LOGS`.

Step 7: Destroy Amazon FSx for NetApp ONTAP and FlexCache

You can optionally delete and remove Amazon FSx for NetApp ONTAP and FlexCache.

1. Set the variable `flexcache_operation` in the `terraform.tfvars` file to "destroy".
2. Navigate to the folder root (AWS_FSXN_BTC) and issue the following command.

```
docker-compose -f docker-compose-destroy.yml up
```

   This command creates two containers. The first container delete FlexCache and the second container deletes FSx for ONTAP.

3. Monitor the provisioning process.

```
docker-compose -f docker-compose-destroy.yml logs -f
```

Amazon FSx for NetApp ONTAP - Disaster recovery

You can use this automation solution to take a disaster recovery backup of a source system using Amazon FSx for NetApp ONTAP.

Amazon FSx for NetApp ONTAP is also referred to as **FSx for ONTAP**.

About this solution

At a high level, the automation code provided with this solution performs the following actions:

- Provision a destination FSx for ONTAP file system
- Provision Storage Virtual Machines (SVMs) for the file system
- Create a cluster peering relationship between the source and destination systems
- Create an SVM peering relationship between the source system and destination system for SnapMirror
- Create destination volumes
- Create a SnapMirror relationship between the source and destination volumes
- Initiate the SnapMirror transfer between the source and destination volumes

The automation is based on Docker and Docker Compose which must be installed on the Linux virtual machine as described below.

Before you begin

You must have the following to complete the provisioning and configuration:

- You need to download the Amazon FSx for NetApp ONTAP - Disaster recovery automation solution through the BlueXP web UI. The solution is packaged as `FSxN_DR.zip`. This zip contains the `AWS_FSxN_Bck_Prov.zip` file that you will use to deploy the solution described in this document.

- Network connectivity between the source and destination systems.
- A Linux VM with the following characteristics:
  - Debian-based Linux distribution
  - Deployed on the same VPC subset used for FSx for ONTAP provisioning
- An AWS account.

Step 1: Install and configure Docker

Install and configure Docker in a Debian-based Linux virtual machine.

Steps

1. Prepare the environment.

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gnupg-
agent softwareproperties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt-get update
```

2. Install Docker and verify the installation.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
docker --version
```

3. Add the required Linux group with an associated user.

   First check if the group **docker** exists in your Linux system. If it doesn't exist, create the group and add the user. By default, the current shell user is added to the group.

```
sudo groupadd docker
sudo usermod -aG docker $(whoami)
```

4. Activate the new group and user definitions

   If you created a new group with a user, you need to activate the definitions. To do this, you can logout of Linux and then back in. Or you can run the following command.

```
newgrp docker
```

Step 2: Install Docker Compose

Install Docker Compose in a Debian-based Linux virtual machine.

Steps

1. Install Docker Compose.

```
sudo curl -L
"https://github.com/docker/compose/releases/latest/download/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

2. Verify the installation was successful.

```
docker-compose --version
```

Step 3: Prepare the Docker image

You need to extract and load the Docker image provided with the automation solution.

Steps

1. Copy the solution file `AWS_FSxN_Bck_Prov.zip` to the virtual machine where the automation code will run.

```
scp -i ~/<private-key.pem> -r AWS_FSxN_Bck_Prov.zip
user@<IP_ADDRESS_OF_VM>
```

   The input parameter `private-key.pem` is your private key file used for AWS virtual machine authentication (EC2 instance).

2. Navigate to the correct folder with the solution file and unzip the file.

```
unzip AWS_FSxN_Bck_Prov.zip
```

3. Navigate to the new folder `AWS_FSxN_Bck_Prov` created with the unzip operation and list the files. You should see file `aws_fsxn_bck_image_latest.tar.gz`.

```
ls -la
```

4. Load the Docker image file. The load operation should normally complete in a few seconds.

```
docker load -i aws_fsxn_bck_image_latest.tar.gz
```

5. Confirm the Docker image is loaded.

```
docker images
```

You should see the Docker image `aws_fsxn_bck_image` with the tag `latest`.

```
   REPOSITORY        TAG     IMAGE ID      CREATED      SIZE
aws_fsxn_bck_image  latest  da87d4974306  2 weeks ago  1.19GB
```

Step 4: Create environment file for AWS credentials

You must create a local variable file for authentication using the access and secret key. Then add the file to the `.env` file.

Steps

1. Create the `awsauth.env` file in the following location:

   `path/to/env-file/awsauth.env`

2. Add the following content to the file:

```
access_key=<>
secret_key=<>
```

   The format **must** be exactly as shown above without any spaces between `key` and `value`.

3. Add the absolute file path to the `.env` file using the `AWS_CREDS` variable. For example:

   `AWS_CREDS=path/to/env-file/awsauth.env`

Step 5: Create an external volume

You need an external volume to make sure the Terraform state files and other important files are persistent. These files must be available for Terraform to run the workflow and deployments.

Steps

1. Create an external volume outside of Docker Compose.

   Make sure to update the volume name (last parameter) to the appropriate value before running the command.

```
docker volume create aws_fsxn_volume
```

2. Add the path to the external volume to the `.env` environment file using the command:

`PERSISTENT_VOL=path/to/external/volume:/volume_name`

Remember to keep the existing file contents and colon formatting. For example:

```
PERSISTENT_VOL=aws_fsxn_volume:/aws_fsxn_bck
```

You can instead add an NFS share as the external volume using a command such as:

`PERSISTENT_VOL=nfs/mnt/document:/aws_fsx_bck`

3. Update the Terraform variables.
   a. Navigate to the folder `aws_fsxn_variables`.
   b. Confirm the following two files exist: `terraform.tfvars` and `variables.tf`.
   c. Update the values in `terraform.tfvars` as required for your environment.

      See Terraform resource: aws_fsx_ontap_file_system for more information.

Step 6: Deploy the backup solution

You can deploy and provision the disaster recovery backup solution.

Steps

1. Navigate to the folder root (AWS_FSxN_Bck_Prov) and issue the provisioning command.

   ```
   docker-compose up -d
   ```

   This command creates three containers. The first container deploys FSx for ONTAP. The second container creates the cluster peering, SVM peering, and destination volume. The third container creates the SnapMirror relationship and initiates the SnapMirror transfer.

2. Monitor the provisioning process.

   ```
   docker-compose logs -f
   ```

   This command gives you the output in real time, but has been configured to capture the logs through the file `deployment.log`. You can change the name of these log files by editing the `.env` file and updating the variables `DEPLOYMENT_LOGS`.

Azure NetApp Files

Install Oracle using Azure NetApp Files

You can use this automation solution to provision Azure NetApp Files volumes and install Oracle on an available virtual machine. Oracle then uses the volumes for data storage.

About this solution

At a high level, the automation code provided with this solution performs the following actions:

- Set up a NetApp account on Azure
- Set up a storage capacity pool on Azure
- Provision the Azure NetApp Files volumes based on the definition
- Create the mount points
- Mount the Azure NetApp Files volumes to the mount points
- Install Oracle on the Linux server
- Create the listeners and database
- Create the Pluggable Databases (PDBs)
- Start the listener and Oracle instance
- Install and configure the `azacsnap` utility to take a snapshot

Before you begin

You must have the following to complete the installation:

- You need to download the Oracle using Azure NetApp Files automation solution through the BlueXP web UI. The solution is packaged as file `na_oracle19c_deploy-master.zip`.
- A Linux VM with the following characteristics:
  - RHEL 8 (Standard_D8s_v3-RHEL-8)
  - Deployed on the same Azure Virtual Network used for the Azure NetApp Files provisioning
- An Azure account

The automation solution is provided as an image and run using Docker and Docker Compose. You need to install both of these on the Linux virtual machine as described below.

You should also register the VM with RedHat using the command `sudo subscription-manager register`. The command will prompt you for your account credentials. If needed, you can create an account at https://developers.redhat.com/.

Step 1: Install and configure Docker

Install and configure Docker in a RHEL 8 Linux virtual machine.

Steps

1. Install the Docker software using the following commands.

```
dnf config-manager --add
-repo=https://download.docker.com/linux/centos/docker-ce.repo
dnf install docker-ce --nobest -y
```

2. Start Docker and display the version to confirm the installation was successful.

```
systemctl start docker
systemctl enable docker
docker --version
```

3. Add the required Linux group with an associated user.

   First check if the group **docker** exists in your Linux system. If it doesn't, create the group and add the user. By default, the current shell user is added to the group.

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

4. Activate the new group and user definitions

   If you created a new group with a user, you need to activate the definitions. To do this, you can logout of Linux and then back in. Or you can run the following command.

```
newgrp docker
```

Step 2: Install Docker Compose and the NFS utilities

Install and configure Docker Compose along with the NFS utilities package.

Steps

1. Install Docker Compose and display the version to confirm the installation was successful.

```
dnf install curl -y
curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

2. Install the NFS utilities package.

```
sudo yum install nfs-utils
```

Step 3: Download the Oracle installation files

Download the required Oracle installation and patch files as well as the `azacsnap` utility.

Steps

1. Sign in to your Oracle account as needed.
2. Download the following files.

File	Description
LINUX.X64_193000_db_home.zip	19.3 base installer
p31281355_190000_Linux-x86-64.zip	19.8 RU patch
p6880880_190000_Linux-x86-64.zip	opatch version 12.2.0.1.23
azacsnap_installer_v5.0.run	azacsnap installer

3. Place all the installation files in the folder `/tmp/archive`.
4. Make sure all users on the database server have full access (read, write, execute) to the folder `/tmp/archive`.

Step 4: Prepare the Docker image

You need to extract and load the Docker image provided with the automation solution.

Steps

1. Copy the solution file `na_oracle19c_deploy-master.zip` to the virtual machine where the automation code will run.

```
scp -i ~/<private-key.pem> -r na_oracle19c_deploy-master.zip
user@<IP_ADDRESS_OF_VM>
```

   The input parameter `private-key.pem` is your private key file used for Azure virtual machine authentication.

2. Navigate to the correct folder with the solution file and unzip the file.

```
unzip na_oracle19c_deploy-master.zip
```

3. Navigate to the new folder `na_oracle19c_deploy-master` created with the unzip operation and list the files. You should see file `ora_anf_bck_image.tar`.

```
ls -lt
```

4. Load the Docker image file. The load operation should normally complete in a few seconds.

```
docker load -i ora_anf_bck_image.tar
```

5. Confirm the Docker image is loaded.

```
docker images
```

You should see the Docker image `ora_anf_bck_image` with the tag `latest`.

```
   REPOSITORY          TAG       IMAGE ID      CREATED      SIZE
ora_anf_bck_image    latest   ay98y7853769   1 week ago   2.58GB
```

Step 5: Create an external volume

You need an external volume to make sure the Terraform state files and other important files are persistent. These files must be available for Terraform to run the workflow and deployments.

Steps

1. Create an external volume outside of Docker Compose.

   Make sure to update the volume name before running the command.

   ```
   docker volume create <VOLUME_NAME>
   ```

2. Add the path to the external volume to the `.env` environment file using the command:

   `PERSISTENT_VOL=path/to/external/volume:/ora_anf_prov`.

   Remember to keep the existing file contents and colon formatting. For example:

   ```
   PERSISTENT_VOL= ora_anf _volume:/ora_anf_prov
   ```

3. Update the Terraform variables.
   a. Navigate to the folder `ora_anf_variables`.
   b. Confirm the following two files exist: `terraform.tfvars` and `variables.tf`.
   c. Update the values in `terraform.tfvars` as required for your environment.

Step 6: Install Oracle

You can now provision and install Oracle.

Steps

1. Install Oracle using the following sequence of commands.

```
docker-compose up terraform_ora_anf
bash /ora_anf_variables/setup.sh
docker-compose up linux_config
bash /ora_anf_variables/permissions.sh
docker-compose up oracle_install
```

2. Reload your Bash variables and confirm by displaying the value for `ORACLE_HOME`.
   a. `cd /home/oracle`
   b. `source .bash_profile`
   c. `echo $ORACLE_HOME`
3. You should be able to login to Oracle.

```
sudo su oracle
```

Step 7: Validate the Oracle installation

You should confirm the Oracle installation was successful.

Steps

1. Log in to the Linux Oracle server and display a list of the Oracle processes. This confirms the installation completed as expected and the Oracle database is running.

```
ps -ef | grep ora
```

2. Log in to the database to examine the database configuration and to confirm the PDBs were created properly.

```
sqlplus / as sysdba
```

   You should see output similar to the following:

```
SQL*Plus: Release 19.0.0.0.0 - Production on Thu May 6 12:52:51 2021
Version 19.8.0.0.0

Copyright (c) 1982, 2019, Oracle. All rights reserved.

Connected to:
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production
Version 19.8.0.0.0
```

3. Execute a few simple SQL commands to confirm the database is available.

```
select name, log_mode from v$database;
show pdbs.
```

Step 8: Install the azacsnap utility and perform a snapshot backup

You need to install and run the `azacsnap` utility to perform a snapshot backup.

Steps

1. Install the container.

```
docker-compose up azacsnap_install
```

2. Switch to the snapshot user account.

```
su - azacsnap
execute /tmp/archive/ora_wallet.sh
```

3. Configure a storage backup detail file. This will create the `azacsnap.json` configuration file.

```
cd /home/azacsnap/bin/
azacsnap -c configure --configuration new
```

4. Perform a snapshot backup.

```
azacsnap -c backup --other data --prefix ora_test --retention=1
```

Step 9: Optionally migrate an on-premise PDB to the cloud

You can optionally migrate the on-premise PDB to the cloud.

Steps

1. Set the variables in the `tfvars` files as needed for your environment.
2. Migrate the PDB.

```
docker-compose -f docker-compose-relocate.yml up
```

Cloud Volumes ONTAP for AWS

Cloud Volumes ONTAP for AWS - Burst to cloud

This article supports the NetApp Cloud Volumes ONTAP for AWS Automation Solution, which is available to NetApp customers from the BlueXP Automation Catalog.

The Cloud Volumes ONTAP for AWS Automation Solution automates the containerized deployment of Cloud Volumes ONTAP for AWS using Terraform, enabling you to deploy Cloud Volumes ONTAP for AWS rapidly, without any manual intervention.

Before you begin

- You must download the Cloud Volumes ONTAP AWS - Burst to cloud automation solution through the BlueXP web UI. The solution is packaged as `cvo_aws_flexcache.zip`.
- You must install a Linux VM on the same network as Cloud Volumes ONTAP.
- After you install the Linux VM, you must follow the steps in this solution to install the required dependencies.

Step 1: Install Docker and Docker Compose

Install Docker

The following steps use Ubuntu 20.04 Debian Linux distribution software as an example. The commands you run depend on the Linux distribution software that you are using. Refer to the specific Linux distribution software documentation for your configuration.

Steps

1. Install Docker by running the following `sudo` commands:

```
sudo apt-get update
sudo apt-get install apt-transport-https cacertificates curl gnupg-agent
software-properties-common curl -fsSL
https://download.docker.com/linux/ubuntu/gpg |
sudo apt-key add -
sudo add-apt-repository “deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable”
sudo apt-get update
sudo apt-get install dockerce docker-ce-cli containerd.io
```

2. Verify the installation:

```
docker –version
```

3. Verify that a group named "docker" has been created on your Linux system. If necessary, create the group:

```
sudo groupadd docker
```

4. Add the user that needs to access Docker to the group:

```
sudo usermod -aG docker $(whoami)
```

5. Your changes are applied after you log out and log back in to the terminal. Alternatively, you can apply the changes immediately:

```
newgrp docker
```

Install Docker Compose

Steps

1. Install Docker Compose by running the following `sudo` commands:

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
```

2. Verify the installation:

```
docker-compose –version
```

Step 2: Prepare the Docker image

Steps

1. Copy the `cvo_aws_flexcache.zip` folder to the Linux VM that you want to use to deploy Cloud Volumes ONTAP:

```
scp -i ~/<private-key>.pem -r cvo_aws_flexcache.zip
<awsuser>@<IP_ADDRESS_OF_VM>:<LOCATION_TO_BE_COPIED>
```

   - `private-key.pem` is your private key file for login without a password.
   - `awsuser` is the VM username.
   - `IP_ADDRESS_OF_VM` is the VM IP address.
   - `LOCATION_TO_BE_COPIED` is the location where the folder will be copied.

2. Extract the `cvo_aws_flexcache.zip` folder. You can extract the folder in the current directory or in a custom location.

   To extract the folder in the current directory, run:

   ```
   unzip cvo_aws_flexcache.zip
   ```

   To extract the folder in a custom location, run:

   ```
   unzip cvo_aws_flexcache.zip -d ~/<your_folder_name>
   ```

3. After you extract the content, navigate to the `CVO_Aws_Deployment` folder and run the following command to view the files:

   ```
    ls -la
   ```

   You should see a list of files, similar to the following example:

   ```
   total 32
       drwxr-xr-x   8 user1  staff   256 Mar 23 12:26 .
       drwxr-xr-x   6 user1  staff   192 Mar 22 08:04 ..
       -rw-r--r--   1 user1  staff   324 Apr 12 21:37 .env
       -rw-r--r--   1 user1  staff  1449 Mar 23 13:19 Dockerfile
       drwxr-xr-x  15 user1  staff   480 Mar 23 13:19 cvo_Aws_source_code
       drwxr-xr-x   4 user1  staff   128 Apr 27 13:43 cvo_Aws_variables
       -rw-r--r--   1 user1  staff   996 Mar 24 04:06 docker-compose-
   deploy.yml
       -rw-r--r--   1 user1  staff  1041 Mar 24 04:06 docker-compose-
   destroy.yml
   ```

4. Locate the `cvo_aws_flexcache_ubuntu_image.tar` file. This contains the Docker image required to deploy Cloud Volumes ONTAP for AWS.
5. Untar the file:

   ```
   docker load -i cvo_aws_flexcache_ubuntu_image.tar
   ```

6. Wait a few minutes for the Docker image to load, and then validate that the Docker image loaded successfully:

   ```
   docker images
   ```

You should see a Docker image named `cvo_aws_flexcache_ubuntu_image` with the `latest` tag, as shown in the following example:

```
REPOSITORY                                TAG         IMAGE ID       CREATED
SIZE
 cvo_aws_flexcache_ubuntu_image     latest    18db15a4d59c   2 weeks ago
1.14GB
```

> You can change the Docker image name if required. If you change the Docker image name, make sure to update the Docker image name in the `docker-compose-deploy` and `docker-compose-destroy` files.

Step 3: Create environment variable files

At this stage, you must create two environment variable files. One file is for authentication of AWS Resource Manager APIs using the AWS access and secret keys. The second file is for setting environment variables to enable BlueXP Terraform modules to locate and authenticate AWS APIs.

Steps

1. Create the `awsauth.env` file in the following location:

   `path/to/env-file/awsauth.env`

   i. Add the following content to the `awsauth.env` file:

   access_key=<>
   secret_key=<>

   The format **must** be exactly as shown above.

2. Add the absolute file path to the `.env` file.

   Enter the absolute path for the `awsauth.env` environment file that corresponds to the `AWS_CREDS` environment variable.

   `AWS_CREDS=path/to/env-file/awsauth.env`

3. Navigate to the `cvo_aws_variable` folder and update the access and secret key in the credentials file.

   Add the following content to the file:

   aws_access_key_id=<>
   aws_secret_access_key=<>

   The format **must** be exactly as shown above.

Step 4: Add Cloud Volumes ONTAP licenses to BlueXP or subscribe to BlueXP

You can add Cloud Volumes ONTAP licenses to BlueXP or subscribe to NetApp BlueXP in the AWS Marketplace.

Steps

1. From the AWS portal, navigate to **SaaS** and select **Subscribe to NetApp BlueXP**.

   You can either use the same resource group as Cloud Volumes ONTAP or a different resource group.

2. Configure the BlueXP portal to import the SaaS subscription to BlueXP.

   You can configure this directly from the AWS portal.

   You are redirected to the BlueXP portal to confirm the configuration.

3. Confirm the configuration in the BlueXP portal by selecting **Save**.

Step 5: Create an external volume

You should create an external volume to keep the Terraform state files, and other important files persistent. You must make sure that the files are available for Terraform to run the workflow and deployments.

Steps

1. Create an external volume outside of Docker Compose:

   ```
   docker volume create <volume_name>
   ```

   Example:

   ```
   docker volume create cvo_aws_volume_dst
   ```

2. Use one of the following options:

   a. Add an external volume path to the `.env` environment file.

      You must follow the exact format shown below.

      Format:

      ```
      PERSISTENT_VOL=path/to/external/volume:/cvo_aws
      ```

      Example:
      `PERSISTENT_VOL=cvo_aws_volume_dst:/cvo_aws`

   b. Add NFS shares as an external volume.

      Make sure that the Docker container can communicate with the NFS shares and that the correct permissions, such as read/write, are configured.

      i. Add the NFS shares path as the path to the external volume in the Docker Compose file, as shown below:
      Format:

      ```
      PERSISTENT_VOL=path/to/nfs/volume:/cvo_aws
      ```

Example:
`PERSISTENT_VOL=nfs/mnt/document:/cvo_aws`

3. Navigate to the `cvo_aws_variables` folder.

   You should see the following variable file in the folder:

   - `terraform.tfvars`
   - `variables.tf`

4. Change the values inside the `terraform.tfvars` file according to your requirements.

   You must read the specific supporting documentation when modifying any of the variable values in the `terraform.tfvars` file. The values can vary depending on region, availability zones, and other factors supported by Cloud Volumes ONTAP for AWS. This includes licenses, disk size, and VM size for single nodes and high availability (HA) pairs.

   All supporting variables for the Connector and Cloud Volumes ONTAP Terraform modules are already defined in the `variables.tf` file. You must refer to the variable names in the `variables.tf` file before adding to the `terraform.tfvars` file.

5. Depending on your requirements, you can enable or disable FlexCache and FlexClone by setting the following options to `true` or `false`.

   The following examples enable FlexCache and FlexClone:

   - `is_flexcache_required = true`
   - `is_flexclone_required = true`

Step 6: Deploy Cloud Volumes ONTAP for AWS

Use the following steps to deploy Cloud Volumes ONTAP for AWS.

Steps

1. From the root folder, run the following command to trigger deployment:

   ```
   docker-compose -f docker-compose-deploy.yml up -d
   ```

   Two containers are triggered, the first container deploys Cloud Volumes ONTAP and the second container sends telemetry data to AutoSupport.

   The second container waits until the first container completes all of the steps successfully.

2. Monitor progress of the deployment process using the log files:

   ```
   docker-compose -f docker-compose-deploy.yml logs -f
   ```

   This command provides output in real time and captures the data in the following log files:
   `deployment.log`

`telemetry_asup.log`

You can change the name of these log files by editing the `.env` file using the following environment variables:

`DEPLOYMENT_LOGS`

`TELEMETRY_ASUP_LOGS`

The following examples show how to change the log file names:

`DEPLOYMENT_LOGS=<your_deployment_log_filename>.log`

`TELEMETRY_ASUP_LOGS=<your_telemetry_asup_log_filename>.log`

After you finish

You can use the following steps to remove the temporary environment and clean up items that were created during the deployment process.

Steps

1. If you deployed FlexCache, set the following option in the `terraform.tfvars` variable file, this cleans up FlexCache volumes and removes the temporary environment that was created earlier.

   `flexcache_operation = "destroy"`

   The possible options are `deploy` and `destroy`.

2. If you deployed FlexClone, set the following option in the `terraform.tfvars` variable file, this cleans up FlexClone volumes and removes the temporary environment that was created earlier.

   `flexclone_operation = "destroy"`

   The possible options are `deploy` and `destroy`.

Cloud Volumes ONTAP for Azure

Cloud Volumes ONTAP for Azure - Burst to cloud

This article supports the NetApp Cloud Volumes ONTAP for Azure Automation Solution, which is available to NetApp customers from the BlueXP Automation Catalog.

The Cloud Volumes ONTAP for Azure Automation Solution automates the containerized deployment of Cloud Volumes ONTAP for Azure using Terraform, enabling you to deploy Cloud Volumes ONTAP for Azure rapidly, without any manual intervention.

Before you begin

- You must download the Cloud Volumes ONTAP Azure - Burst to cloud automation solution through the BlueXP web UI. The solution is packaged as `CVO-Azure-Burst-To-Cloud.zip`.
- You must install a Linux VM on the same network as Cloud Volumes ONTAP.

- After you install the Linux VM, you must follow the steps in this solution to install the required dependencies.

Step 1: Install Docker and Docker Compose

Install Docker

The following steps use Ubuntu 20.04 Debian Linux distribution software as an example. The commands you run depend on the Linux distribution software that you are using. Refer to the specific Linux distribution software documentation for your configuration.

Steps

1. Install Docker by running the following `sudo` commands:

```
sudo apt-get update
sudo apt-get install apt-transport-https cacertificates curl gnupg-agent
software-properties-common curl -fsSL
https://download.docker.com/linux/ubuntu/gpg |
sudo apt-key add -
sudo add-apt-repository “deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable”
sudo apt-get update
sudo apt-get install dockerce docker-ce-cli containerd.io
```

2. Verify the installation:

```
docker –version
```

3. Verify that a group named "docker" has been created on your Linux system. If necessary, create the group:

```
sudo groupadd docker
```

4. Add the user that needs to access Docker to the group:

```
sudo usermod –aG docker $(whoami)
```

5. Your changes are applied after you log out and log back in to the terminal. Alternatively, you can apply the changes immediately:

```
newgrp docker
```

Install Docker Compose

Steps

1. Install Docker Compose by running the following `sudo` commands:

```
sudo curl -L
“https://github.com/docker/compose/releases/download/1.29.2/dockercompos
e-(□□□□□ - □)-(uname -m)” -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

2. Verify the installation:

```
docker-compose –version
```

Step 2: Prepare the Docker image

Steps

1. Copy the `CVO-Azure-Burst-To-Cloud.zip` folder to the Linux VM that you want to use to deploy Cloud Volumes ONTAP:

```
scp -i ~/<private-key>.pem -r CVO-Azure-Burst-To-Cloud.zip
<azureuser>@<IP_ADDRESS_OF_VM>:<LOCATION_TO_BE_COPIED>
```

   - `private-key.pem` is your private key file for login without a password.
   - `azureuser` is the VM username.
   - `IP_ADDRESS_OF_VM` is the VM IP address.
   - `LOCATION_TO_BE_COPIED` is the location where the folder will be copied.

2. Extract the `CVO-Azure-Burst-To-Cloud.zip` folder. You can extract the folder in the current directory or in a custom location.

   To extract the folder in the current directory, run:

```
unzip CVO-Azure-Burst-To-Cloud.zip
```

   To extract the folder in a custom location, run:

```
unzip CVO-Azure-Burst-To-Cloud.zip -d ~/<your_folder_name>
```

3. After you extract the content, navigate to the `CVO_Azure_Deployment` folder and run the following command to view the files:

```
ls -la
```

You should see a list of files, similar to the following example:

```
drwxr-xr-x@ 11 user1 staff 352 May 5 13:56 .
drwxr-xr-x@ 5 user1 staff 160 May 5 14:24 ..
-rw-r--r--@ 1 user1 staff 324 May 5 13:18 .env
-rw-r--r--@ 1 user1 staff 1449 May 5 13:18 Dockerfile
-rw-r--r--@ 1 user1 staff 35149 May 5 13:18 LICENSE
-rw-r--r--@ 1 user1 staff 13356 May 5 14:26 README.md
-rw-r--r-- 1  user1 staff 354318151 May 5 13:51
cvo_azure_flexcache_ubuntu_image_latest
drwxr-xr-x@ 4 user1 staff 128 May 5 13:18 cvo_azure_variables
-rw-r--r--@ 1 user1 staff 996 May 5 13:18 docker-compose-deploy.yml
-rw-r--r--@ 1 user1 staff 1041 May 5 13:18 docker-compose-destroy.yml
-rw-r--r--@ 1 user1 staff 4771 May 5 13:18 sp_role.json
```

4. Locate the `cvo_azure_flexcache_ubuntu_image_latest.tar.gz` file. This contains the Docker image required to deploy Cloud Volumes ONTAP for Azure.
5. Untar the file:

```
docker load -i cvo_azure_flexcache_ubuntu_image_latest.tar.gz
```

6. Wait a few minutes for the Docker image to load, and then validate that the Docker image loaded successfully:

```
docker images
```

You should see a Docker image named `cvo_azure_flexcache_ubuntu_image_latest` with the `latest` tag, as shown in the following example:

```
REPOSITORY TAG IMAGE ID CREATED SIZE
cvo_azure_flexcache_ubuntu_image latest 18db15a4d59c 2 weeks ago 1.14GB
```

Step 3: Create environment variable files

At this stage, you must create two environment variable files. One file is for authentication of Azure Resource Manager APIs using service principal credentials. The second file is for setting environment variables to enable BlueXP Terraform modules to locate and authenticate Azure APIs.

Steps

1. Create a service principal.

Before you can create the environment variable files, you must create a service principal by following the steps in Create an Azure Active Directory application and service principal that can access resources.

2. Assign the **Contributor** role to the newly created service principal.
3. Create a custom role.
   a. Locate the `sp_role.json` file and check for the required permissions under the actions listed.
   b. Insert these permissions and attach the custom role to the newly created service principal.
4. Navigate to **Certificates & secrets** and select **New client secret** to create the client secret.

   When you create the client secret, you must record the details from the **Value** column because you will not be able to see this value again. You must also record the following information:

   - Client ID
   - Subscription ID
   - Tenant ID

     You will need this information to create the environment variables. You can find client ID and tenant ID information in the **Overview** section of the Service Principal UI.

5. Create the environment files.
   a. Create the `azureauth.env` file in the following location:

      `path/to/env-file/azureauth.env`

      i. Add the following content to the file:

         clientId=<> clientSecret=<> subscriptionId=<> tenantId=<>

         The format **must** be exactly as shown above without any spaces between the key and value.

   b. Create the `credentials.env` file in the following location:

      `path/to/env-file/credentials.env`

      i. Add the following content to the file:

         AZURE_TENANT_ID=<> AZURE_CLIENT_SECRET=<>
         AZURE_CLIENT_ID=<> AZURE_SUBSCRIPTION_ID=<>

         The format **must** be exactly as shown above without any spaces between the key and value.

6. Add the absolute file paths to the `.env` file.

   Enter the absolute path for the `azureauth.env` environment file in the `.env` file that corresponds to the `AZURE_RM_CREDS` environment variable.

   `AZURE_RM_CREDS=path/to/env-file/azureauth.env`

   Enter the absolute path for the `credentials.env` environment file in the `.env` file that corresponds to the `BLUEXP_TF_AZURE_CREDS` environment variable.

```
BLUEXP_TF_AZURE_CREDS=path/to/env-file/credentials.env
```

Step 4: Add Cloud Volumes ONTAP licenses to BlueXP or subscribe to BlueXP

You can add Cloud Volumes ONTAP licenses to BlueXP or subscribe to NetApp BlueXP in the Azure Marketplace.

Steps

1. From the Azure portal, navigate to **SaaS** and select **Subscribe to NetApp BlueXP**.
2. Select the **Cloud Manager (by Cap PYGO by Hour, WORM and data services)** plan.

   You can either use the same resource group as Cloud Volumes ONTAP or a different resource group.

3. Configure the BlueXP portal to import the SaaS subscription to BlueXP.

   You can configure this directly from the Azure portal by navigating to **Product and plan details** and selecting the **Configure account now** option.

   You will then be redirected to the BlueXP portal to confirm the configuration.

4. Confirm the configuration in the BlueXP portal by selecting **Save**.

Step 5: Create an external volume

You should create an external volume to keep the Terraform state files, and other important files persistent. You must make sure that the files are available for Terraform to run the workflow and deployments.

Steps

1. Create an external volume outside of Docker Compose:

   ```
   docker volume create « volume_name »
   ```

   Example:

   ```
   docker volume create cvo_azure_volume_dst
   ```

2. Use one of the following options:

   a. Add an external volume path to the `.env` environment file.

      You must follow the exact format shown below.

      Format:

      ```
      PERSISTENT_VOL=path/to/external/volume:/cvo_azure
      ```

      Example:
      `PERSISTENT_VOL=cvo_azure_volume_dst:/cvo_azure`

   b. Add NFS shares as an external volume.

Make sure that the Docker container can communicate with the NFS shares and that the correct permissions, such as read/write, are configured.

i. Add the NFS shares path as the path to the external volume in the Docker Compose file, as shown below:
Format:

```
PERSISTENT_VOL=path/to/nfs/volume:/cvo_azure
```

Example:
`PERSISTENT_VOL=nfs/mnt/document:/cvo_azure`

3. Navigate to the `cvo_azure_variables` folder.

You should see the following variable files in the folder:

`terraform.tfvars`

`variables.tf`

4. Change the values inside the `terraform.tfvars` file according to your requirements.

You must read the specific supporting documentation when modifying any of the variable values in the `terraform.tfvars` file. The values can vary depending on region, availability zones and other factors supported by Cloud Volumes ONTAP for Azure. This includes licenses, disk size, and VM size for single nodes and high availability (HA) pairs.

All supporting variables for the Connector and Cloud Volumes ONTAP Terraform modules are already defined in the `variables.tf` file. You must refer to the variable names in the `variables.tf` file before adding to the `terraform.tfvars` file.

5. Depending on your requirements, you can enable or disable FlexCache and FlexClone by setting the following options to `true` or `false`.

The following examples enable FlexCache and FlexClone:

- `is_flexcache_required = true`
- `is_flexclone_required = true`

6. If necessary, you can retrieve the value for the Terraform `az_service_principal_object_id` variable from the Azure Active Directory Service:
   a. Navigate to **Enterprise Applications –> All Applications** and select the name of the Service Principal you created earlier.
   b. Copy the object ID and insert the value for the Terraform variable:

`az_service_principal_object_id`

Step 6: Deploy Cloud Volumes ONTAP for Azure

Use the following steps to deploy Cloud Volumes ONTAP for Azure.

Steps

1. From the root folder, run the following command to trigger deployment:

```
docker-compose up -d
```

   Two containers are triggered, the first container deploys Cloud Volumes ONTAP and the second container sends telemetry data to AutoSupport.

   The second container waits until the first container completes all of the steps successfully.

2. Monitor progress of the deployment process using the log files:

```
docker-compose logs -f
```

   This command provides output in real time and captures the data in the following log files:

   `deployment.log`

   `telemetry_asup.log`

   You can change the name of these log files by editing the `.env` file using the following environment variables:

   `DEPLOYMENT_LOGS`

   `TELEMETRY_ASUP_LOGS`

   The following examples show how to change the log file names:

   `DEPLOYMENT_LOGS=<your_deployment_log_filename>.log`

   `TELEMETRY_ASUP_LOGS=<your_telemetry_asup_log_filename>.log`

After you finish

You can use the following steps to remove the temporary environment and clean up items that were created during the deployment process.

Steps

1. If you deployed FlexCache, set the following option in the `terraform.tfvars` file, this cleans up FlexCache volumes and removes the temporary environment that was created earlier.

   `flexcache_operation = "destroy"`

   The possible options are `deploy` and `destroy`.

2. If you deployed FlexClone, set the following option in the `terraform.tfvars` file, this cleans up FlexClone volumes and removes the temporary environment that was created earlier.

   `flexclone_operation = "destroy"`

The possible options are `deploy` and `destroy`.

Cloud Volumes ONTAP for Google Cloud

Cloud Volumes ONTAP for Google Cloud - Burst to cloud

This article supports the NetApp Cloud Volumes ONTAP for Google Cloud Automation Solution, which is available to NetApp customers from the BlueXP Automation Catalog.

The Cloud Volumes ONTAP for Google Cloud Automation Solution automates the containerized deployment of Cloud Volumes ONTAP for Google Cloud, enabling you to deploy Cloud Volumes ONTAP for Google Cloud rapidly, without any manual intervention.

Before you begin

- You must download the Cloud Volumes ONTAP for Google Cloud - Burst to cloud automation solution through the BlueXP web UI. The solution is packaged as `cvo_gcp_flexcache.zip`.
- You must install a Linux VM on the same network as Cloud Volumes ONTAP.
- After you install the Linux VM, you must follow the steps in this solution to install the required dependencies.

Step 1: Install Docker and Docker Compose

Install Docker

The following steps use Ubuntu 20.04 Debian Linux distribution software as an example. The commands you run depend on the Linux distribution software that you are using. Refer to the specific Linux distribution software documentation for your configuration.

Steps

1. Install Docker by running the following commands:

```
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gnupg-
agent software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

2. Verify the installation:

```
docker –version
```

3. Verify that a group named "docker" has been created on your Linux system. If necessary, create the group:

```
sudo groupadd docker
```

4. Add the user that needs to access Docker to the group:

```
sudo usermod -aG docker $(whoami)
```

5. Your changes are applied after you log out and log back in to the terminal. Alternatively, you can apply the changes immediately:

```
newgrp docker
```

Install Docker Compose

Steps

1. Install Docker Compose by running the following `sudo` commands:

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose
```

2. Verify the installation:

```
docker-compose –version
```

Step 2: Prepare the Docker image

Steps

1. Copy the `cvo_gcp_flexcache.zip` folder to the Linux VM that you want to use to deploy Cloud Volumes ONTAP:

```
scp -i ~/private-key.pem -r cvo_gcp_flexcache.zip
gcpuser@IP_ADDRESS_OF_VM:LOCATION_TO_BE_COPIED
```

   - `private-key.pem` is your private key file for login without a password.
   - `gcpuser` is the VM username.
   - `IP_ADDRESS_OF_VM` is the VM IP address.

- `LOCATION_TO_BE_COPIED` is the location where the folder will be copied.

2. Extract the `cvo_gcp_flexcache.zip` folder. You can extract the folder in the current directory or in a custom location.

   To extract the folder in the current directory, run:

   ```
   unzip cvo_gcp_flexcache.zip
   ```

   To extract the folder in a custom location, run:

   ```
   unzip cvo_gcp_flexcache.zip -d ~/<your_folder_name>
   ```

3. After you extract the content, run the following command to view the files:

   ```
    ls -la
   ```

   You should see a list of files, similar to the following example:

   ```
       total 32
       drwxr-xr-x   8 user  staff   256 Mar 23 12:26 .
       drwxr-xr-x   6 user  staff   192 Mar 22 08:04 ..
       -rw-r--r--   1 user  staff   324 Apr 12 21:37 .env
       -rw-r--r--   1 user  staff  1449 Mar 23 13:19 Dockerfile
       drwxr-xr-x  15 user  staff   480 Mar 23 13:19 cvo_gcp_source_code
       drwxr-xr-x   4 user  staff   128 Apr 27 13:43 cvo_gcp_variables
       -rw-r--r--   1 user  staff   996 Mar 24 04:06 docker-compose-
   deploy.yml
       -rw-r--r--   1 user  staff  1041 Mar 24 04:06 docker-compose-
   destroy.yml
   ```

4. Locate the `cvo_gcp_flexcache_ubuntu_image.tar` file. This contains the Docker image required to deploy Cloud Volumes ONTAP for Google Cloud.
5. Untar the file:

   ```
   docker load -i cvo_gcp_flexcache_ubuntu_image.tar
   ```

6. Wait a few minutes for the Docker image to load, and then validate that the Docker image loaded successfully:

   ```
   docker images
   ```

You should see a Docker image named `cvo_gcp_flexcache_ubuntu_image` with the `latest` tag, as shown in the following example:

```
REPOSITORY                                TAG         IMAGE ID       CREATED
SIZE
    cvo_gcp_flexcache_ubuntu_image     latest    18db15a4d59c   2 weeks
ago   1.14GB
```

> You can change the Docker image name if required. If you change the Docker image name, make sure to update the Docker image name in the `docker-compose-deploy` and `docker-compose-destroy` files.

Step 3: Update the JSON file

At this stage, you must update the `cxo-automation-gcp.json` file with a service account key to authenticate the Google Cloud provider.

1. Create a service account with permissions to deploy Cloud Volumes ONTAP and the BlueXP Connector. Learn more about creating service accounts.
2. Download the key file for the account and update the `cxo-automation-gcp.json` file with the key file information. The `cxo-automation-gcp.json` file is located in the `cvo_gcp_variables` folder.

Example

```
{
  "type": "service_account",
  "project_id": "",
  "private_key_id": "",
  "private_key": "",
  "client_email": "",
  "client_id": "",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url":
"https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "",
  "universe_domain": "googleapis.com"
}
```

The file format must be exactly as shown above.

Step 4: Subscribe to BlueXP

You can subscribe to NetApp BlueXP in the Google Cloud Marketplace.

Steps

1. Navigate to the Google Cloud console and select **Subscribe to NetApp BlueXP**.
2. Configure the BlueXP portal to import the SaaS subscription to BlueXP.

   You can configure this directly from the Google Cloud Platform. You will be redirected to the BlueXP portal to confirm the configuration.

3. Confirm the configuration in the BlueXP portal by selecting **Save**.

For more information, see Manage Google Cloud credentials and subscriptions for BlueXP.

Step 5: Enable required Google Cloud APIs

You must enable the following Google Cloud APIs in your project to deploy Cloud Volumes ONTAP and the Connector.

- Cloud Deployment Manager V2 API
- Cloud Logging API
- Cloud Resource Manager API
- Compute Engine API
- Identity and Access Management (IAM) API

Learn more about enabling APIs

Step 6: Create an external volume

You should create an external volume to keep the Terraform state files and other important files persistent. You must make sure that the files are available for Terraform to run the workflow and deployments.

Steps

1. Create an external volume outside of Docker Compose:

   ```
   docker volume create <volume_name>
   ```

   Example:

   ```
   docker volume create cvo_gcp_volume_dst
   ```

2. Use one of the following options:

   a. Add an external volume path to the `.env` environment file.

      You must follow the exact format shown below.

      Format:

      `PERSISTENT_VOL=path/to/external/volume:/cvo_gcp`

      Example:
      `PERSISTENT_VOL=cvo_gcp_volume_dst:/cvo_gcp`

b. Add NFS shares as an external volume.

   Make sure that the Docker container can communicate with the NFS shares and that the correct permissions, such as read/write, are configured.

   i. Add the NFS shares path as the path to the external volume in the Docker Compose file, as shown below:
   Format:

   ```
   PERSISTENT_VOL=path/to/nfs/volume:/cvo_gcp
   ```

   Example:
   `PERSISTENT_VOL=nfs/mnt/document:/cvo_gcp`

3. Navigate to the `cvo_gcp_variables` folder.

   You should see the following files in the folder:

   - `terraform.tfvars`
   - `variables.tf`

4. Change the values inside the `terraform.tfvars` file according to your requirements.

   You must read the specific supporting documentation when modifying any of the variable values in the `terraform.tfvars` file. The values can vary depending on region, availability zones, and other factors supported by Cloud Volumes ONTAP for Google Cloud. This includes licenses, disk size, and VM size for single nodes and high availability (HA) pairs.

   All supporting variables for the Connector and Cloud Volumes ONTAP Terraform modules are already defined in the `variables.tf` file. You must refer to the variable names in the `variables.tf` file before adding to the `terraform.tfvars` file.

5. Depending on your requirements, you can enable or disable FlexCache and FlexClone by setting the following options to `true` or `false`.

   The following examples enable FlexCache and FlexClone:

   - `is_flexcache_required = true`
   - `is_flexclone_required = true`

Step 7: Deploy Cloud Volumes ONTAP for Google Cloud

Use the following steps to deploy Cloud Volumes ONTAP for Google Cloud.

Steps

1. From the root folder, run the following command to trigger deployment:

   ```
   docker-compose -f docker-compose-deploy.yml up -d
   ```

   Two containers are triggered, the first container deploys Cloud Volumes ONTAP and the second container sends telemetry data to AutoSupport.

The second container waits until the first container completes all of the steps successfully.

2. Monitor progress of the deployment process using the log files:

```
docker-compose -f docker-compose-deploy.yml logs -f
```

This command provides output in real time and captures the data in the following log files:
`deployment.log`

`telemetry_asup.log`

You can change the name of these log files by editing the `.env` file using the following environment variables:

`DEPLOYMENT_LOGS`

`TELEMETRY_ASUP_LOGS`

The following examples show how to change the log file names:

`DEPLOYMENT_LOGS=<your_deployment_log_filename>.log`

`TELEMETRY_ASUP_LOGS=<your_telemetry_asup_log_filename>.log`

After you finish

You can use the following steps to remove the temporary environment and clean up items that were created during the deployment process.

Steps

1. If you deployed FlexCache, set the following option in the `terraform.tfvars` file, this cleans up FlexCache volumes and removes the temporary environment that was created earlier.

```
flexcache_operation = "destroy"
```

> The possible options are `deploy` and `destroy`.

2. If you deployed FlexClone, set the following option in the `terraform.tfvars` file, this cleans up FlexClone volumes and removes the temporary environment that was created earlier.

```
flexclone_operation = "destroy"
```

> The possible options are `deploy` and `destroy`.

ONTAP

Day 0/1

Overview of the ONTAP day 0/1 solution

You can use the ONTAP day 0/1 automation solution to deploy and configure an ONTAP cluster using Ansible. The solution is available from the BlueXP automation catalog.

Flexible ONTAP deployment options

Depending on your requirements, you can use on-premises hardware or Simulate ONTAP to deploy and configure an ONTAP cluster using Ansible.

On-premises hardware

You can deploy this solution using on-premises hardware running ONTAP, such as a FAS or an AFF system. You must use a Linux VM to deploy and configure the ONTAP cluster using Ansible.

Simulate ONTAP

To deploy this solution using an ONTAP simulator, you must download the latest version of Simulate ONTAP from the NetApp support site. Simulate ONTAP is a virtual simulator for ONTAP software. Simulate ONTAP runs in a VMware hypervisor on a Windows, Linux, or Mac system. For Windows and Linux hosts, you must use the VMware Workstation hypervisor to run this solution. If you have a Mac OS, use the VMware Fusion hypervisor.

Layered design

The Ansible framework simplifies the development and reuse of automation execution and logic tasks. The framework makes a distinction between decision-making tasks (logic layer), and the execution steps (execution layer) in automation. Understanding how these layers work enables you to customize the configuration.

An Ansible "playbook" executes a series of tasks from start to finish. The `site.yml` playbook contains the `logic.yml` playbook and the `execution.yml` playbook.

When a request is run, the `site.yml` playbook makes a call to the `logic.yml` playbook first, and then calls the `execution.yml` playbook to execute the service request.

You are not required to use the logic layer of the framework. The logic layer provides options to expand the capability of the framework beyond the hard-coded values for execution. This enables you to customize the framework capabilities if required.

Logic layer

The logic layer consists of the following:

- The `logic.yml` playbook
- Logic task files within the `logic-tasks` directory

The logic layer provides the capability for complex decision making without the need for significant custom integration (for example, connecting to ServiceNOW). The logic layer is configurable and provides the input to microservices.

The ability to bypass the logic layer is also provided. If you want to bypass the logic layer, do not define the `logic_operation` variable. Direct invocation of the `logic.yml` playbook provides the ability to do some level of debugging without execution. You can use a "debug" statement to verify that the value of the `raw_service_request` is correct.

Important considerations:

- The `logic.yml` playbook searches for the `logic_operation` variable. If the variable is defined in the request, it loads a task file from the `logic-tasks` directory. The task file must be a .yml file. If there is no matching task file and the `logic_operation` variable is defined, the logic layer fails.
- The default value of the `logic_operation` variable is `no-op`. If the variable is not explicitly defined, it defaults to `no-op`, which does not run any operations.
- If the `raw_service_request` variable is already defined, then execution proceeds to the execution layer. If the variable is not defined, the logic layer fails.

Execution layer

The execution layer consists of the following:

- The `execution.yml` playbook

The execution layer makes the API calls to configure an ONTAP cluster. The `execution.yml` playbook requires that the `raw_service_request` variable is defined when executed.

Support for customization

You can customize this solution in various ways depending on your requirements.

Customization options include:

- Modifying Ansible playbooks
- Adding roles

Customize Ansible files

The following table describes the customizable Ansible files contained in this solution.

Location	Description
`playbooks/inventory/hosts`	Contains a single file with a list of hosts and groups.
`playbooks/group_vars/all/*`	Ansible provides a convenient way to apply variables to multiple hosts at once. You can modify any or all files in this this folder including `cfg.yml`, `clusters.yml`, `defaults.yml`, `services.yml`, `standards.yml`, and `vault.yml`.
`playbooks/logic-tasks`	Supports decision-making tasks within Ansible and maintains the separation of logic and execution. You can add files to this folder that correspond to the relevant service.
`playbooks/vars/*`	Dynamic values used within Ansible playbooks and roles to enable customization, flexibility, and reusability of configurations. If necessary, you can modify any or all files in this folder.

Customize roles

You can also customize the solution by adding or changing Ansible roles, also called microservices. For more details, see Customize.

Prepare to use the ONTAP day 0/1 solution

Before you deploy the automation solution, you must prepare the ONTAP environment and install and configure Ansible.

Initial planning considerations

You should review the following requirements and considerations before using this solution to deploy an ONTAP cluster.

Basic requirements

You must meet the following basic requirements to use this solution:

- You must have access to ONTAP software, either on-premises or through an ONTAP simulator.
- You must know how to use ONTAP software.
- You must know how to use Ansible automation software tools.

Planning considerations

Before deploying this automation solution, you must decide:

- The location where you are going to run the Ansible control node.
- The ONTAP system, either on-premises hardware or an ONTAP simulator.
- Whether or not you will require customization.

Prepare the ONTAP system

Whether you are using an on-premises ONTAP system or Simulate ONTAP, you must prepare the environment before you can deploy the automation solution.

Optionally, install and configure Simulate ONTAP

If you want to deploy this solution through an ONTAP simulator, you must download and run Simulate ONTAP.

Before you begin

- You must download and install the VMware hypervisor that you are going to use to run Simulate ONTAP.
  - If you have a Windows or Linux OS, use VMware Workstation.
  - If you have a Mac OS, use VMware Fusion.

    If you are using a Mac OS, you must have an Intel processor.

Steps

Use the following procedure to install two ONTAP simulators in your local environment:

1. Download Simulate ONTAP from the NetApp support site.

   Although you install two ONTAP simulators, you only need to download one copy of the software.

2. If it is not already running, start your VMware application.

3. Locate the simulator file that was downloaded and right click to open it with the VMware application.
4. Set the name of the first ONTAP instance.
5. Wait for the simulator boot up and follow the directions to create a single node cluster.

   Repeat the steps for the second ONTAP instance.

6. Optionally, add a full disk complement.

   From each cluster, run the following commands:

```
security unlock -username <user_01>
security login password -username <user_01>
set -priv advanced
systemshell local
disk assign -all -node <Cluster-01>-01
```

State of the ONTAP system

You must verify the initial state of the ONTAP system, whether it is on-premises or running through an ONTAP simulator.

Verify that the following ONTAP system requirements are met:

- ONTAP is installed and running with no cluster defined yet.
- ONTAP is booted and displaying the IP address to access the cluster.
- The network is reachable.
- You have admin credentials.
- The Message of the Day (MOTD) banner is displayed with the management address.

Install the required automation software

This section provides information on how to install Ansible and prepare the automation solution for deployment.

Install Ansible

Ansible can be installed on Linux or Windows systems.

The default communication method that Ansible uses to communicate with an ONTAP cluster is SSH.

Refer to Getting Started with NetApp and Ansible: Install Ansible to install Ansible.

Ansible must be installed on the control node of the system.

Download and prepare the automation solution

You can use the following steps to download and prepare the automation solution for deployment.

1. Download the ONTAP - Day 0/1 & Health Checks automation solution through the BlueXP web UI. The solution is packaged as `ONTAP_DAY0_DAY1.zip`.

2. Extract the zip folder and copy the files to the desired location on the control node within your Ansible environment.

Initial Ansible framework configuration

Perform the initial configuration of the Ansible framework:

1. Navigate to `playbooks/inventory/group_vars/all`.
2. Decrypt the `vault.yml` file:

   ```
   ansible-vault decrypt playbooks/inventory/group_vars/all/vault.yml
   ```

   When prompted for the vault password, enter the following temporary password:

   ```
   NetApp123!
   ```

   > "NetApp123!" is a temporary password to decrypt the `vault.yml` file and the corresponding vault password. After first use, you **must** encrypt the file using your own password.

3. Modify the following Ansible files:
   - `clusters.yml` - Modify the values in this file to suit your environment.
   - `vault.yml` - After decrypting the file, modify the ONTAP cluster, username and password values to suit your environment.
   - `cfg.yml` - Set the file path for `log2file` and set `show_request` under `cfg` to `True` to display the `raw_service_request`.

     The `raw_service_request` variable is displayed in the log files and during execution.

     > Each file listed contains comments with instructions on how to modify it according to your requirements.

4. Re-encrypt the `vault.yml` file:

   ```
   ansible-vault encrypt playbooks/inventory/group_vars/all/vault.yml
   ```

   > You are prompted to choose a new password for the vault upon encryption.

5. Navigate to `playbooks/inventory/hosts` and set a valid Python interpreter.
6. Deploy the `framework_test` service:

   The following command runs the `na_ontap_info` module with a `gather_subset` value of `cluster_identity_info`. This validates that the basic configuration is correct and verifies that you can communicate with the cluster.

```
ansible-playbook -i inventory/hosts site.yml -e
cluster_name=<CLUSTER_NAME>
-e logic_operation=framework-test
```

Run the command for each cluster.

If successful, you should see output similar to the following example:

```
PLAY RECAP
*********************************************************************
*********
localhost : ok=12 changed=1 unreachable=0 failed=0 skipped=6
The key is 'rescued=0' and 'failed=0'..
```

Deploy the ONTAP cluster using the solution

After completing the preparation and planning, you are ready to use the ONTAP day 0/1 solution to quickly configure an ONTAP cluster using Ansible.

At any time during the steps in this section, you can choose to test a request instead of actually executing it. To test a request, change the `site.yml` playbook on the command line to `logic.yml`.

> The `docs/tutorial-requests.txt` location contains the final version of all service requests used throughout this procedure. If you have difficulty running a service request, you can copy the relevant request from the `tutorial-requests.txt` file to the `playbooks/inventory/group_vars/all/tutorial-requests.yml` location and modify the hard-coded values as required (IP address, aggregate names and so on). You should then be able to successfully run the request.

Before you begin

- You must have Ansible installed.
- You must have downloaded the ONTAP day 0/1 solution and extracted the folder to the desired location on the Ansible control node.
- The ONTAP system state must meet the requirements and you must have the necessary credentials.
- You must have completed all required tasks outlined in the Prepare section.

> The examples throughout this solution use "Cluster_01" and "Cluster_02" as the names for the two clusters. You must replace these values with the names of the clusters in your environment.

Step 1: Initial cluster configuration

At this stage, you must perform some initial cluster configuration steps.

Steps

1. Navigate to the `playbooks/inventory/group_vars/all/tutorial-requests.yml` location and

review the `cluster_initial` request in the file. Make any necessary changes for your environment.

2. Create a file in the `logic-tasks` folder for the service request. For example, create a file called `cluster_initial.yml`.

   Copy the following lines to the new file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial cluster configuration
  set_fact:
    raw_service_request:
```

3. Define the `raw_service_request` variable.

   You can use one of the following options to define the `raw_service_request` variable in the `cluster_initial.yml` file you created in the `logic-tasks` folder:

   - **Option 1**: Manually define the `raw_service_request` variable.

     Open the `tutorial-requests.yml` file using an editor and copy the content from line 11 to line 165. Paste the content under the `raw service request` variable in the new `cluster_initial.yml` file, as shown in the following examples:


3 # This file contains the final version of the various service
4 # requests used throughout the tutorial in TUTORIAL.md.
5 #---
6 #---
7 # cluster_initial:
8 #
9 #---
11 service: cluster_initial
create
13 std_name: none
14 req_details:
15
16 ontap_aggr:
17 - hostname: "{{ cluster_name }}"
18 disk_count: 24
19 name: n01_aggr1
20 nodes: "{{ cluster_name }}-01"

Show example

Example `cluster_initial.yml` file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial cluster configuration
  set_fact:
    raw_service_request:
     service:          cluster_initial
     operation:         create
     std_name:           none
     req_details:

      ontap_aggr:
      - hostname:                   "{{ cluster_name }}"
        disk_count:                 24
        name:                       n01_aggr1
        nodes:                      "{{ cluster_name }}-01"
        raid_type:                  raid4

      - hostname:                   "{{ peer_cluster_name }}"
        disk_count:                 24
        name:                       n01_aggr1
        nodes:                      "{{ peer_cluster_name }}-01"
        raid_type:                  raid4

      ontap_license:
      - hostname:                   "{{ cluster_name }}"
        license_codes:
        - XXXXXXXXXXXXXXXAAAAAAAAAAAAAA
        - XXXXXXXXXXXXXXXAAAAAAAAAAAAAA
```

```
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA

  - hostname:                   "{{ peer_cluster_name }}"
    license_codes:
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
```

```
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA
      - XXXXXXXXXXXXXXAAAAAAAAAAAAAA

  ontap_motd:
  - hostname:                     "{{ cluster_name }}"
    vserver:                      "{{ cluster_name }}"
    message:                      "New MOTD"

  - hostname:                     "{{ peer_cluster_name }}"
    vserver:                      "{{ peer_cluster_name }}"
    message:                      "New MOTD"

  ontap_interface:
  - hostname:                     "{{ cluster_name }}"
    vserver:                      "{{ cluster_name }}"
    interface_name:               ic01
    role:                         intercluster
    address:                      10.0.0.101
    netmask:                      255.255.255.0
    home_node:                    "{{ cluster_name }}-01"
    home_port:                    e0c
    ipspace:                      Default
    use_rest:                     never

  - hostname:                     "{{ cluster_name }}"
    vserver:                      "{{ cluster_name }}"
    interface_name:               ic02
    role:                         intercluster
    address:                      10.0.0.101
    netmask:                      255.255.255.0
    home_node:                    "{{ cluster_name }}-01"
    home_port:                    e0c
```

```
    ipspace:                    Default
    use_rest:                   never

  - hostname:                   "{{ peer_cluster_name }}"
    vserver:                    "{{ peer_cluster_name }}"
    interface_name:             ic01
    role:                       intercluster
    address:                    10.0.0.101
    netmask:                    255.255.255.0
    home_node:                  "{{ peer_cluster_name }}-01"
    home_port:                  e0c
    ipspace:                    Default
    use_rest:                   never

  - hostname:                   "{{ peer_cluster_name }}"
    vserver:                    "{{ peer_cluster_name }}"
    interface_name:             ic02
    role:                       intercluster
    address:                    10.0.0.101
    netmask:                    255.255.255.0
    home_node:                  "{{ peer_cluster_name }}-01"
    home_port:                  e0c
    ipspace:                    Default
    use_rest:                   never

  ontap_cluster_peer:
  - hostname:                   "{{ cluster_name }}"
    dest_cluster_name:          "{{ peer_cluster_name }}"
    dest_intercluster_lifs:     "{{ peer_lifs }}"
    source_cluster_name:        "{{ cluster_name }}"
    source_intercluster_lifs:   "{{ cluster_lifs }}"
    peer_options:
      hostname:                 "{{ peer_cluster_name }}"
```

- **Option 2**: Use a Jinja template to define the request:

  You can also use the following Jinja template format to get the `raw_service_request` value.

  ```
  raw_service_request: "{{ cluster_initial }}"
  ```

4. Perform the initial cluster configuration for the first cluster:

```
ansible-playbook -i inventory/hosts site.yml -e
cluster_name=<Cluster_01>
```

Verify that there are no errors before proceeding.

5. Repeat the command for the second cluster:

```
ansible-playbook -i inventory/hosts site.yml -e
cluster_name=<Cluster_02>
```

Verify that there are no errors for the second cluster.

When you scroll up towards the beginning of the Ansible output you should see the request that was sent to the framework, as shown in the following example:

Show example

```
TASK [Show the raw_service_request]
********************************************************************
****************************************
ok: [localhost] => {
    "raw_service_request": {
        "operation": "create",
        "req_details": {
            "ontap_aggr": [
                {
                    "disk_count": 24,
                    "hostname": "Cluster_01",
                    "name": "n01_aggr1",
                    "nodes": "Cluster_01-01",
                    "raid_type": "raid4"
                }
            ],
            "ontap_license": [
                {
                    "hostname": "Cluster_01",
                    "license_codes": [
                        "XXXXXXXXXXXXXXXAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
```

```
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA",
                        "XXXXXXXXXXXXXXAAAAAAAAAAAAA"
                    ]
                }
            ],
            "ontap_motd": [
                {
                    "hostname": "Cluster_01",
                    "message": "New MOTD",
                    "vserver": "Cluster_01"
                }
            ]
        },
        "service": "cluster_initial",
        "std_name": "none"
    }
}
```

6. Log in to each ONTAP instance and verify that the request was successful.

Step 2: Configure the intercluster LIFs

You can now configure the intercluster LIFs by adding the LIF definitions to the `cluster_initial` request and defining the `ontap_interface` microservice.

The service definition and the request work together to determine the action:

- If you provide a service request for a microservice that is not in the service definitions, the request is not executed.
- If you provide a service request with one or more microservices defined in the service definitions, but omitted from the request, the request is not executed.

The `execution.yml` playbook evaluates the service definition by scanning the list of microservices in the order listed:

- If there is an entry in the request with a dictionary key matching the `args` entry contained in the microservice definitions, the request is executed.
- If there is no matching entry in the service request, the request is skipped without error.

Steps

1. Navigate to the `cluster_initial.yml` file that you created previously and modify the request by adding the following lines to the request definitions:

```
ontap_interface:
- hostname:                    "{{ cluster_name }}"
  vserver:                     "{{ cluster_name }}"
  interface_name:              ic01
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never

- hostname:                    "{{ cluster_name }}"
  vserver:                     "{{ cluster_name }}"
  interface_name:              ic02
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never

- hostname:                    "{{ peer_cluster_name }}"
  vserver:                     "{{ peer_cluster_name }}"
  interface_name:              ic01
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ peer_cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never

- hostname:                    "{{ peer_cluster_name }}"
  vserver:                     "{{ peer_cluster_name }}"
  interface_name:              ic02
  role:                        intercluster
  address:                     <ip_address>
  netmask:                     <netmask_address>
  home_node:                   "{{ peer_cluster_name }}-01"
  home_port:                   e0c
  ipspace:                     Default
  use_rest:                    never
```

2. Run the command:

```
ansible-playbook -i inventory/hosts  site.yml -e
cluster_name=<Cluster_01> -e peer_cluster_name=<Cluster_02>
```

3. Log in to each instance to check if the LIFs have been added to the cluster:

Show example

```
Cluster_01::> net int show
  (network interface show)
           Logical    Status     Network            Current
Current Is
Vserver     Interface  Admin/Oper Address/Mask       Node
Port    Home
----------- ---------- ---------- ------------------ -------------
------- ----
Cluster_01
            Cluster_01-01_mgmt up/up 10.0.0.101/24   Cluster_01-01
e0c     true
            Cluster_01-01_mgmt_auto up/up 10.101.101.101/24
Cluster_01-01 e0c true
            cluster_mgmt up/up    10.0.0.110/24      Cluster_01-01
e0c     true
5 entries were displayed.
```

The output shows that the LIFs were **not** added. This is because the `ontap_interface` microservice still needs to be defined in the `services.yml` file.

4. Verify that the LIFs were added to the `raw_service_request` variable.

Show example

The following example shows that the LIFs have been added to the request:

```
            "ontap_interface": [
                 {
                     "address": "10.0.0.101",
                     "home_node": "Cluster_01-01",
                     "home_port": "e0c",
                     "hostname": "Cluster_01",
                     "interface_name": "ic01",
                     "ipspace": "Default",
                     "netmask": "255.255.255.0",
                     "role": "intercluster",
                     "use_rest": "never",
                     "vserver": "Cluster_01"
                 },
                 {
                     "address": "10.0.0.101",
                     "home_node": "Cluster_01-01",
                     "home_port": "e0c",
                     "hostname": "Cluster_01",
                     "interface_name": "ic02",
                     "ipspace": "Default",
                     "netmask": "255.255.255.0",
                     "role": "intercluster",
                     "use_rest": "never",
                     "vserver": "Cluster_01"
                 },
                 {
                     "address": "10.0.0.101",
                     "home_node": "Cluster_02-01",
                     "home_port": "e0c",
                     "hostname": "Cluster_02",
                     "interface_name": "ic01",
                     "ipspace": "Default",
                     "netmask": "255.255.255.0",
                     "role": "intercluster",
                     "use_rest": "never",
                     "vserver": "Cluster_02"
                 },
                 {
                     "address": "10.0.0.126",
                     "home_node": "Cluster_02-01",
                     "home_port": "e0c",
                     "hostname": "Cluster_02",
```

```
                    "interface_name": "ic02",
                    "ipspace": "Default",
                    "netmask": "255.255.255.0",
                    "role": "intercluster",
                    "use_rest": "never",
                    "vserver": "Cluster_02"
                }
            ],
```

5. Define the `ontap_interface` microservice under `cluster_initial` in the `services.yml` file.

   Copy the following lines to the file to define the microservice:

```
        - name: ontap_interface
          args: ontap_interface
          role: na/ontap_interface
```

6. Now that the `ontap_interface` microservice has been defined in the request and the `services.yml` file, run the request again:

```
ansible-playbook -i inventory/hosts  site.yml -e
cluster_name=<Cluster_01> -e peer_cluster_name=<Cluster_02>
```

7. Log in to each ONTAP instance and verify that the LIFs have been added.

Step 3: Optionally, configure multiple clusters

If required, you can configure multiple clusters in the same request. You must provide variable names for each cluster when you define the request.

Steps

1. Add an entry for the second cluster in the `cluster_initial.yml` file to configure both clusters in the same request.

   The following example displays the `ontap_aggr` field after the second entry is added.

```
ontap_aggr:
 - hostname:                    "{{ cluster_name }}"
   disk_count:                  24
   name:                        n01_aggr1
   nodes:                       "{{ cluster_name }}-01"
   raid_type:                   raid4

 - hostname:                    "{{ peer_cluster_name }}"
   disk_count:                  24
   name:                        n01_aggr1
   nodes:                       "{{ peer_cluster_name }}-01"
   raid_type:                   raid4
```

2. Apply the changes for all other items under `cluster_initial`.
3. Add cluster peering to the request by copying the following lines to the file:

```
ontap_cluster_peer:
- hostname:                     "{{ cluster_name }}"
  dest_cluster_name:            "{{ cluster_peer }}"
  dest_intercluster_lifs:       "{{ peer_lifs }}"
  source_cluster_name:          "{{ cluster_name }}"
  source_intercluster_lifs:     "{{ cluster_lifs }}"
  peer_options:
    hostname:                   "{{ cluster_peer }}"
```

4. Run the Ansible request:

```
ansible-playbook -i inventory/hosts -e cluster_name=<Cluster_01>
site.yml -e peer_cluster_name=<Cluster_02> -e
cluster_lifs=<cluster_lif_1_IP_address,cluster_lif_2_IP_address>
-e peer_lifs=<peer_lif_1_IP_address,peer_lif_2_IP_address>
```

Step 4: Initial SVM configuration

At this stage in the procedure, you configure the SVMs in the cluster.

Steps

1. Update the `svm_initial` request in the `tutorial-requests.yml` file to configure an SVM and SVM peer relationship.

   You must configure the following:

   - The SVM

- The SVM peer relationship
- The SVM interface for each SVM

2. Update the variable definitions in the `svm_initial` request definitions. You must modify the following variable definitions:
   - `cluster_name`
   - `vserver_name`
   - `peer_cluster_name`
   - `peer_vserver`

   To update the definitions, remove the **'{}'** after `req_details` for the `svm_initial` definition and add the correct definition.

3. Create a file in the `logic-tasks` folder for the service request. For example, create a file called `svm_initial.yml`.

   Copy the following lines to the file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial SVM configuration
  set_fact:
    raw_service_request:
```

4. Define the `raw_service_request` variable.

   You can use one of the following options to define the `raw_service_request` variable for `svm_initial` in the `logic-tasks` folder:

   - **Option 1**: Manually define the `raw_service_request` variable.

     Open the `tutorial-requests.yml` file using an editor and copy the content from line 179 to line

222. Paste the content under the `raw service request` variable in the new `svm_initial.yml` file, as shown in the following examples:

Show example

Example `svm_initial.yml` file:

```
- name: Validate required inputs
  ansible.builtin.assert:
    that:
    - service is defined

- name: Include data files
  ansible.builtin.include_vars:
    file:   "{{ data_file_name }}.yml"
  loop:
  - common-site-stds
  - user-inputs
  - cluster-platform-stds
  - vserver-common-stds
  loop_control:
    loop_var:    data_file_name

- name: Initial SVM configuration
  set_fact:
    raw_service_request:
     service:          svm_initial
     operation:        create
     std_name:         none
     req_details:

      ontap_vserver:
      - hostname:                   "{{ cluster_name }}"
        name:                       "{{ vserver_name }}"
        root_volume_aggregate:      n01_aggr1

      - hostname:                   "{{ peer_cluster_name }}"
       name:                       "{{ peer_vserver }}"
       root_volume_aggregate:      n01_aggr1

      ontap_vserver_peer:
      - hostname:                   "{{ cluster_name }}"
        vserver:                    "{{ vserver_name }}"
        peer_vserver:               "{{ peer_vserver }}"
        applications:               snapmirror
        peer_options:
          hostname:                 "{{ peer_cluster_name }}"

      ontap_interface:
```

```
    - hostname:                    "{{ cluster_name }}"
      vserver:                     "{{ vserver_name }}"
      interface_name:              data01
      role:                        data
      address:                     10.0.0.200
      netmask:                     255.255.255.0
      home_node:                   "{{ cluster_name }}-01"
      home_port:                   e0c
      ipspace:                     Default
      use_rest:                    never

    - hostname:                    "{{ peer_cluster_name }}"
      vserver:                     "{{ peer_vserver }}"
      interface_name:              data01
      role:                        data
      address:                     10.0.0.201
      netmask:                     255.255.255.0
      home_node:                   "{{ peer_cluster_name }}-01"
      home_port:                   e0c
      ipspace:                     Default
      use_rest:                    never
```

- **Option 2**: Use a Jinja template to define the request:

  You can also use the following Jinja template format to get the `raw_service_request` value.

```
raw_service_request: "{{ svm_initial }}"
```

5. Run the request:

```
ansible-playbook -i inventory/hosts -e cluster_name=<Cluster_01> -e
peer_cluster_name=<Cluster_02> -e peer_vserver=<SVM_02>  -e
vserver_name=<SVM_01> site.yml
```

6. Log in to each ONTAP instance and validate the configuration.
7. Add the SVM interfaces.

   Define the `ontap_interface` service under `svm_initial` in the `services.yml` file and run the request again:

```
ansible-playbook -i inventory/hosts -e cluster_name=<Cluster_01> -e
peer_cluster_name=<Cluster_02> -e peer_vserver=<SVM_02>  -e
vserver_name=<SVM_01> site.yml
```

8. Log in to each ONTAP instance and verify that the SVM interfaces have been configured.

Step 5: Optionally, define a service request dynamically

In the previous steps, the `raw_service_request` variable is hard-coded. This is useful for learning, development, and testing. You can also dynamically generate a service request.

The following section provides an option to dynamically produce the required `raw_service_request` if you do not want to integrate it with higher level systems.

- If the `logic_operation` variable is not defined in the command, the `logic.yml` file does not import any file from the `logic-tasks` folder. This means the `raw_service_request` must be defined outside of Ansible and provided to the framework on execution.
- A task file name in the `logic-tasks` folder must match the value of the `logic_operation` variable without the .yml extension.
- The task files in the `logic-tasks` folder dynamically define a `raw_service_request`. The only requirement is that a valid `raw_service_request` be defined as the last task in the relevant file.

How to dynamically define a service request

There are multiple ways to apply a logic task to dynamically define a service request. Some of these options are listed below:

- Using a Ansible task file from the `logic-tasks` folder
- Invoking a custom role that returns data suitable for converting to a `raw_service_request` varaible.
- Invoking another tool outside of the Ansible environment to provide the required data. For example, a REST API call to Active IQ Unified Manager.

The following example commands dynamically define a service request for each cluster using the `tutorial-requests.yml` file:

```
ansible-playbook -i inventory/hosts -e cluster2provision=Cluster_01
-e logic_operation=tutorial-requests site.yml
```

```
ansible-playbook -i inventory/hosts -e cluster2provision=Cluster_02
-e logic_operation=tutorial-requests site.yml
```

Step 6: Deploy the ONTAP day 0/1 solution

At this stage you should have already completed the following:

- Reviewed and modified all files in `playbooks/inventory/group_vars/all` according to your requirements. There are detailed comments in each file to help you make the changes.
- Added any required task files to the the `logic-tasks` directory.
- Added any required data files to the `playbook/vars` directory.

Use the following commands to deploy the ONTAP day 0/1 solution and verify the health of your deployment:

> At this stage, you should have already decrypted and modified the `vault.yml` file and it must be encrypted with your new password.

- Run the ONTAP day 0 service:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=cluster_day_0 -e service=cluster_day_0 -vvvv --ask-vault
-pass <your_vault_password>
```

- Run the ONTAP day 1 service:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=cluster_day_1 -e service=cluster_day_0 -vvvv --ask-vault
-pass <your_vault_password>
```

- Apply cluster wide settings:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=cluster_wide_settings -e service=cluster_wide_settings
-vvvv --ask-vault-pass <your_vault_password>
```

- Run health checks:

```
ansible-playbook -i playbooks/inventory/hosts playbooks/site.yml -e
logic_operation=health_checks -e service=health_checks -e
enable_health_reports=true -vvvv --ask-vault-pass <your_vault_password>
```

Customize the ONTAP day 0/1 solution

To customize the ONTAP day 0/1 solution for your requirements, you can add or change Ansible roles.

Roles represent the microservices within the Ansible framework. Each microservice performs one operation. For example, ONTAP day 0 is a service that contains multiple microservices.

Add Ansible roles

You can add Ansible roles to customize the solution for your environment. Required roles are defined by service definitions within the Ansible framework.

A role must meet the following requirements to be used as a microservice:

- Accept a list of arguments in the `args` variable.
- Use the Ansible "block, rescue, always" structure with certain requirements for each block.
- Use a single Ansible module and define a single task within the block.
- Implement every available module parameter according to the requirements detailed in this section.

Required microservice structure

Each role must support the following variables:

- `mode`: If mode is set to `test` the role attempts to import the `test.yml` which shows what the role does without actually executing it.

  It is not always possible to implement this because of certain interdependencies.

- `status`: The overall status of playbook execution. If the value is not set to `success` the role is not executed.
- `args` : A list of role specific dictionaries with keys that match the role parameter names.
- `global_log_messages`: Gathers log messages during playbook execution. There is one entry generated each time the role is executed.
- `log_name`: The name used to refer to the role within the `global_log_messages` entries.
- `task_descr`: A brief description of what the role does.
- `service_start_time`: The timestamp used to track the time each role is executed.
- `playbook_status`: The status of the Ansible playbook.
- `role_result`: The variable that contains role output and is included in each message within the `global_log_messages` entries.

Example role structure

The following example provides the basic structure of a role that implements a microservice. You must change the variables in this example for your configuration.

Show example

Basic role structure:

```
- name:  Set some role attributes
  set_fact:
    log_name:     "<LOG_NAME>"
    task_descr:   "<TASK_DESCRIPTION>"

-  name: "{{ log_name }}"
   block:
      -  set_fact:
            service_start_time: "{{ lookup('pipe', 'date
+%Y%m%d%H%M%S') }}"

      -  name: "Provision the new user"
         <MODULE_NAME>:

#-----------------------------------------------------------------
            # COMMON ATTRIBUTES

#-----------------------------------------------------------------
            hostname:            "{{
clusters[loop_arg['hostname']]['mgmt_ip'] }}"
            username:            "{{
clusters[loop_arg['hostname']]['username'] }}"
            password:            "{{
clusters[loop_arg['hostname']]['password'] }}"

            cert_filepath:       "{{ loop_arg['cert_filepath']
| default(omit) }}"
            feature_flags:       "{{ loop_arg['feature_flags']
| default(omit) }}"
            http_port:           "{{ loop_arg['http_port']
| default(omit) }}"
            https:               "{{ loop_arg['https']
| default('true') }}"
            ontapi:              "{{ loop_arg['ontapi']
| default(omit) }}"
            key_filepath:        "{{ loop_arg['key_filepath']
| default(omit) }}"
            use_rest:            "{{ loop_arg['use_rest']
| default(omit) }}"
            validate_certs:      "{{ loop_arg['validate_certs']
| default('false') }}"
```

```
            <MODULE_SPECIFIC_PARAMETERS>

#-------------------------------------------------------------
            # REQUIRED ATTRIBUTES

#-------------------------------------------------------------
            required_parameter:     "{{ loop_arg['required_parameter']
}}"

#-------------------------------------------------------------
            # ATTRIBUTES w/ DEFAULTS

#-------------------------------------------------------------
            defaulted_parameter:    "{{ loop_arg['defaulted_parameter']
| default('default_value') }}"

#-------------------------------------------------------------
            # OPTIONAL ATTRIBUTES

#-------------------------------------------------------------
            optional_parameter:     "{{ loop_arg['optional_parameter']
| default(omit) }}"
         loop:    "{{ args }}"
         loop_control:
            loop_var:   loop_arg
         register:   role_result

   rescue:
      -  name: Set role status to FAIL
         set_fact:
            playbook_status:   "failed"

   always:
      -  name: add log msg
         vars:
            role_log:
               role: "{{ log_name }}"
               timestamp:
                  start_time: "{{service_start_time}}"
                  end_time: "{{ lookup('pipe', 'date +%Y-%m-
%d@%H:%M:%S') }}"
               service_status: "{{ playbook_status }}"
               result: "{{role_result}}"
         set_fact:
            global_log_msgs:   "{{ global_log_msgs + [ role_log ] }}"
```

Variables used in the example role:

- `<NAME>`: A replaceable value that must be provided for each microservice.
- `<LOG_NAME>`: The short form name of the role used for logging purposes. For example, `ONTAP_VOLUME`.
- `<TASK_DESCRIPTION>`: A brief description of the what the microservice does.
- `<MODULE_NAME>`: The Ansible module name for the task.

> The top level `execute.yml` playbook specifies the `netapp.ontap` collection. If the module is part of the `netapp.ontap` collection, there is no need to fully specify the module name.

- `<MODULE_SPECIFIC_PARAMETERS>`: Ansible module parameters that are specific to the module used to implement the microservice. The following list describes types of parameters and how they should be grouped.
  - Required parameters: All required parameters are specified with no default value.
  - Parameters that have a default value specific to the microservice (not the same as a default value specified by the module documentation).
  - All remaining parameters use `default(omit)` as the default value.

Using multi-level dictionaries as module parameters

Some NetApp provided Ansible modules use multi-level dictionaries for module parameters (for example, fixed and adaptive QoS policy groups).

Using `default(omit)` alone does not work when these dictionaries are used, especially when there is more than one and they are mutually exclusive.

If you need to use multi-level dictionaries as module parameters, you should split the functionality into multiple microservices (roles) so that each one is guaranteed to supply at least one second-level dictionary value for the relevant dictionary.

The following examples show fixed and adaptive QoS policy groups split across two microservices.

The first microservice contains fixed QoS policy group values:

```
fixed_qos_options:
  capacity_shared:           "{{
loop_arg['fixed_qos_options']['capacity_shared']         | default(omit)
}}"
  max_throughput_iops:       "{{
loop_arg['fixed_qos_options']['max_throughput_iops']     | default(omit)
}}"
  min_throughput_iops:       "{{
loop_arg['fixed_qos_options']['min_throughput_iops']     | default(omit)
}}"
  max_throughput_mbps:       "{{
loop_arg['fixed_qos_options']['max_throughput_mbps']     | default(omit)
}}"
  min_throughput_mbps:       "{{
loop_arg['fixed_qos_options']['min_throughput_mbps']     | default(omit)
}}"
```

The second microservice contains the adaptive QoS policy group values:

```
adaptive_qos_options:
  absolute_min_iops:         "{{
loop_arg['adaptive_qos_options']['absolute_min_iops'] | default(omit) }}"
  expected_iops:             "{{
loop_arg['adaptive_qos_options']['expected_iops']     | default(omit) }}"
  peak_iops:                 "{{
loop_arg['adaptive_qos_options']['peak_iops']         | default(omit) }}"
```

Copyright information

Trademark information