SoftWare Repository for Container

Best Practices

Issue 01

Date 2025-09-26

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Security Declaration

Vulnerability

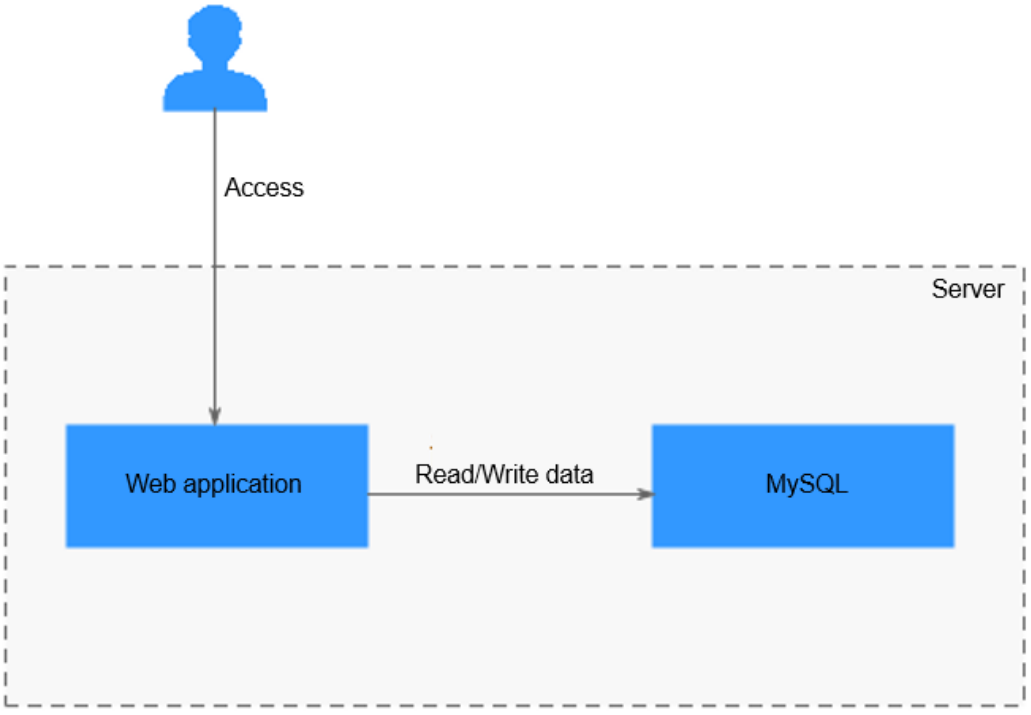
Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process.* For details about this process, visit the following web page:
https://www.huawei.com/en/psirt/vul-response-process
For vulnerability information, enterprise customers can visit the following web page:
https://securitybulletin.huawei.com/enterprise/en/security-advisory

Contents

1 Writing a Quality Dockerfile

This document walks you through how to compile an efficient Dockerfile, using the containerization of an application as an example. Based on the practices of SWR, this file exemplifies how to create images of fewer layers and smaller size to speed up image build process.

The following figure shows a common architecture of an enterprise portal website. This website consists of a web server that provides web services, and a database that stores user data. Normally, the website is deployed on a single server.



To containerize the application, a Dockerfile may be written as follows:

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y nodejs ssh mysql
```



```
RUN cd /app && npm install

# this should start three processes, mysql and ssh
# in the background and node app in foreground
# isn't it beautifully terrible? <3
CMD mysql & sshd & npm start
```

However, the preceding Dockerfile, including the **CMD** command, is problematic.

To rectify and optimize the Dockerfile, here are some tips:

- **Run Only One Process in Each Container**
- **Do Not Upgrade the Tag During Image Build**
- **Merge RUN Commands of Similar Update Frequency**
- **Specify an Image Tag**
- **Delete Unnecessary Files**
- **Select a Suitable Base Image**
- **Set WORKDIR and CMD**
- **(Optional) Use ENTRYPOINT**
- **Use exec in ENTRYPOINT**
- **Use COPY Preferentially**
- **Change the Order of COPY and RUN**
- **Set Default Environment Variables, Mapping Ports, and Data Volumes**
- **Use EXPOSE to Set Listening Ports**
- **Use VOLUME to Manage Data Volumes**
- **Use Labels to Configure Image Metadata**
- **Use HEALTHCHECK**
- **Compile the .dockerignore File**

Run Only One Process in Each Container

Technically, multiple processes, including database, frontend, backend, and SSH, can run on the same Docker container. However, this is not what containers are built for. Stuffing all the processes into one container not only makes the image extremely large in size, but also prolongs the container building time and wastes resources when you perform horizontal scaling. This is because the whole container has to be rebuilt every time you make small adjustments and the number of containers for each application can only be equally added during scaling in.

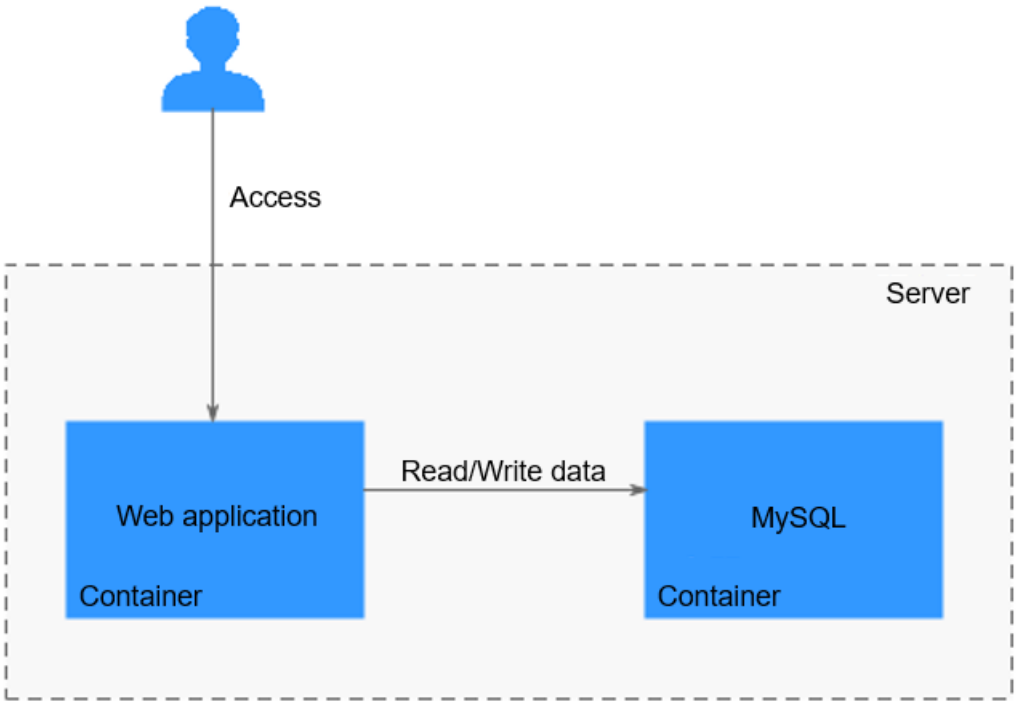
Therefore, usually, an application is split into microservices before containerization. You will benefit a lot from the microservice architecture:

- **Independent scaling**: After an application is split into independent microservices, you can adjust the number of pods for each microservice separately.
- **Faster development**: Since microservices are decoupled, they can be coded independently from each other.
- **Security assurance through isolation**: For an overall application, if a security vulnerability exists, attackers can use this vulnerability to obtain the permission for all functions of the application. However, in a microservice

architecture, if a service is attacked, attackers can only obtain the access permission for this service, but cannot intrude other services.

- **Stabler service**: If one microservice breaks down, other microservices can still run properly.

To optimize the preceding sample Dockerfile, run the web application and MySQL in different containers.



You can modify them separately. As shown in the following example, MySQL is deleted from the sample Dockerfile. Only Node.js is installed.

```
FROM ubuntu

ADD . /app

RUN apt-get update
RUN apt-get upgrade -y

RUN apt-get install -y nodejs
RUN cd /app && npm install

CMD npm start
```

Do Not Upgrade the Tag During Image Build

To reduce image complexity, dependency, size, and build time, do not install any unnecessary packages in your images. For example, do not include a text editor in a database image.

Contact the package maintenance personnel if a package in the base image is out of date but you do not know which package it is. To upgrade a specific package automatically, for example, **foo**, run the **apt-get install -y foo** command.

apt-get upgrade brings great uncertainty to image build. Inconsistency between images might occur as you are not sure what packages have been installed by

apt-get upgrade during image build. Therefore, **apt-get upgrade** is usually deleted.

The following is the sample Dockerfile without **apt-get upgrade**:

```
FROM ubuntu

ADD . /app

RUN apt-get update

RUN apt-get install -y nodejs
RUN cd /app && npm install

CMD npm start
```

Merge RUN Commands of Similar Update Frequency

Like an onion, a Docker image consists of many layers. To modify an inner layer, you need to delete all outer layers. Docker images have the following features:

- Each command in a Dockerfile creates an image layer.
- Image layers are cached and reused.
- Cached image layers expire when the files they copy or variables specified in image build change.
- When a cached image layer expires, its subsequent cached image layers expire accordingly.
- Image layers are immutable. If a file is added into a layer and then deleted in the next layer, the file still exists in the image. The file just turns unavailable in the Docker container.

Therefore, merge multiple commands that are of similar updating probability to avoid unnecessary costs. In the sample Dockerfile, **Node.js** and **npm** are installed together. That means **Node.js** is reinstalled each time the source code is modified, which is time and resource consuming.

```
FROM ubuntu

ADD . /app

RUN apt-get update \
   && apt-get install -y nodejs \
   && cd /app \
   && npm install

CMD npm start
```

It would be better to write the Dockerfile as follows:

```
FROM ubuntu

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```


Specify an Image Tag

If no tag is specified for an image, the image will be tagged with **latest** by default. For example, the **FROM ubuntu** command is equivalent to **FROM ubuntu:latest**. During an image update, **latest** will point to a new tag. The image build may fail.

In the sample Dockerfile, tag the **ubuntu** image with **16.04** as follows:

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y nodejs
ADD . /app
RUN cd /app && npm install

CMD npm start
```

Delete Unnecessary Files

Assume that you have updated the **apt-get** sources, installed some software packages, and saved them in the **/var/lib/apt/lists/** directory.

However, these files are not required to run applications. To make the Docker image more lightweight, it is advised to delete these unnecessary files.

Therefore, in the sample Dockerfile, the files in the **/var/lib/apt/lists/** directory are deleted.

```
FROM ubuntu:16.04

RUN apt-get update \
    && apt-get install -y nodejs \
    && rm -rf /var/lib/apt/lists/*

ADD . /app
RUN cd /app && npm install

CMD npm start
```

Select a Suitable Base Image

In our sample Dockerfile, **ubuntu** is selected as the base image. However, as you only need to run a node program, there is no need to use a general base image. A **node** image would be a better choice.

A **node** image tagged with **alpine** is recommended. Alpine is a lightweight Linux distribution with a size of only 4 MB.

```
FROM node:7-alpine

ADD . /app
RUN cd /app && npm install

CMD npm start
```

Set WORKDIR and CMD

WORKDIR can be used to set a default directory where **RUN**, **CMD**, and **ENTRYPOINT** commands will be run.

CMD provides default commands to be executed when running a container from an image. Write the commands in an array.


```
FROM node:7-alpine

WORKDIR /app
ADD . /app
RUN npm install

CMD ["npm", "start"]
```

(Optional) Use ENTRYPOINT

ENTRYPOINT is optional because it increases complexity. **ENTRYPOINT** is a script that is executed by default. It uses the specified commands as its parameters. It is usually used to create executable Docker images.

```
FROM node:7-alpine

WORKDIR /app
ADD . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Use exec in ENTRYPOINT

In the preceding **ENTRYPOINT** script, **exec** is used to run a node application. If **exec** is not used in **ENTRYPOINT**, the container cannot be successfully closed because the **SIGTERM** signal is interrupted by the **bash** process. The process started by **exec** can replace the **bash** process. In this way, all signals can work normally.

Use COPY Preferentially

COPY is simply used to copy files to images. **ADD** is more complex and can be used to download remote files and decompress packages.

```
FROM node:7-alpine

WORKDIR /app

COPY . /app
RUN npm install

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Change the Order of COPY and RUN

Place the parts that are infrequently changed in the front of your Dockerfile to make the most out of the image cache.

In the sample Dockerfile, the source code changes frequently. Every time the image is built, npm needs to be reinstalled. To avoid this issue, copy **package.json** first, then install npm, and at last copy the rest of the source code. In this way, changes of the source code will not result in repetitive installation of npm.

```
FROM node:7-alpine

WORKDIR /app

COPY package.json /app
RUN npm install
```



```
COPY . /app

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Set Default Environment Variables, Mapping Ports, and Data Volumes

Environment variables may be required when running a Docker container. Setting default environment variables in Dockerfile is a good choice. In addition, you can set mapping ports and data volumes in the Dockerfile. Example:

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Environment variables specified by **ENV** can be used in containers. If you only need to specify variables for image build, you can use **ARG** instead.

Use EXPOSE to Set Listening Ports

EXPOSE is used to describe which ports your containers will listen on. For example, set **EXPOSE 80** for an Apache image and **EXPOSE 27017** for a MongoDB image.

For external access, use a flag to map ports when executing **docker run**.

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV APP_PORT=3000
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Use VOLUME to Manage Data Volumes

VOLUME is used to access database storage files, configuration files, or files and directories of created containers. You are advised to use **VOLUME** to manage the image modules that can change or the modules modifiable for users.

In the sample Dockerfile, a media directory is added.

```
FROM node:7-alpine

ENV PROJECT_DIR=/app

WORKDIR $PROJECT_DIR
```



```
COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```

Use Labels to Configure Image Metadata

Add labels to help organize images, record permissions, and automate image build. Starting with **LABEL**, add one or more labels with each label occupying one line.

NOTICE

If your string contains spaces, put the string in quotation marks ("") or convert the spaces into escape characters. If the string itself contains quotation marks, convert the quotation marks.

```
FROM node:7-alpine
LABEL com.example.version="0.0.1-beta"
```

Use HEALTHCHECK

When running a container, you can enable the **--restart always** option. In this case, the Docker daemon restarts the container when the container crashes. This option is useful for containers that need to run for a long time. What if a container is running but unavailable? **HEALTHCHECK** enables Docker to periodically check the health status of containers. You only need to specify a command. If the containers are normal, **0** is returned. Otherwise, **1** is returned. When the request fails and the **curl --fail** command is run, a non-zero state is returned. Example:

```
FROM node:7-alpine
LABEL com.example.version="0.0.1-beta"

ENV PROJECT_DIR=/app
WORKDIR $PROJECT_DIR

COPY package.json $PROJECT_DIR
RUN npm install
COPY . $PROJECT_DIR

ENV MEDIA_DIR=/media \
    APP_PORT=3000

VOLUME $MEDIA_DIR
EXPOSE $APP_PORT
HEALTHCHECK CMD curl --fail http://localhost:$APP_PORT || exit 1

ENTRYPOINT ["./entrypoint.sh"]
CMD ["start"]
```


Compile the .dockerignore File

The functions and syntax of the **.dockerignore** file are similar to those of the **.gitignore** file. You can ignore unnecessary files to accelerate image build and reduce image size.

Before image build, Docker needs to prepare the context by collecting all required files to the process. By default, the context contains all files in the Dockerfile directory. However, files in directories such as **.git** are unnecessary.

Example:

```
.git/
```


2 Creating a JDK8 Base Image and Pushing It to SWR

Scenarios

A base image can be used as a basis to create other images. This section describes how to create a JDK8 base image on a CCE node and push it to SWR.

Procedure

Step 1 Buy a CCE cluster.

1. Log in to the **CCE console**.
2. On the CCE cluster purchase page, configure parameters. For details, see **Creating a Kubernetes Cluster**.
3. Wait until the cluster is created. The new cluster will be displayed in the cluster list. The cluster status is running and the number of nodes is 0.

Step 2 Create a CCE node.

After a cluster is created, you need to create nodes for running workloads in the cluster. Assume that the node you are creating has Linux and Docker installed. You can use it to create a base image.

In the following steps, CentOS 7.6 is used as an example to describe how to create a JDK8 base image and push it to SWR.

1. Log in to the **CCE console**.
2. Click the cluster created in **Step 1**.
3. In the navigation pane, choose **Nodes**. On the **Nodes** tab, click **Create Node** in the upper right corner. In the displayed dialog box, **configure node parameters**.
4. For network settings, select **Auto create** for **EIP**. Set the bandwidth to **5 Mbit/s**.
5. Click **Next: Confirm**.
6. Check the node specifications, read the instructions, select **I have read and agree to CCE Disclaimer**, and click **Submit**.

Wait until the node is created. The new node will be displayed in the node list. The node status is **Running**.

Step 3 Download a JDK package.

1. After a node is created, click the node name to go to its details page.
2. In the upper right corner, click **Remote Login**.
3. Select a login mode and click **Log In**.
4. Log in to the node as **root**.
5. Create a directory **image**.

   mkdir image
6. Go to the **image** directory.

   cd image
7. Download a JDK package.

   wget https://builds.openlogic.com/downloadJDK/openlogic-openjdk/8u352-b08/openlogic-openjdk-8u352-b08-linux-x64.tar.gz

Step 4 Build an image.

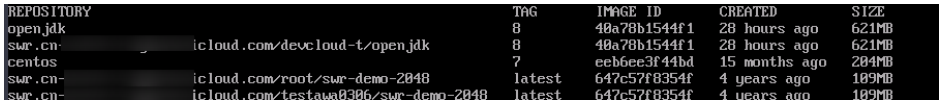
1. Run **vi dockerfile** to write a Dockerfile as follows:

```
FROM centos                                                                    # Use CentOS as the base image.
RUN useradd -d /home/springboot -m springboot                                  # Create a user in the working directory.
ADD ./openlogic-openjdk-8u352-b08-linux-x64.tar.gz /home/springboot            # Add the JDK software package to the image and decompress it automatically.
RUN chown springboot:springboot /home/springboot/openlogic-openjdk-8u352-b08-linux-x64 -R
USER springboot                                                                # Set the user to springboot.
ENV  JAVA_HOME=/home/springboot/openlogic-openjdk-8u352-b08-linux-x64           # Set environment variables.
ENV   PATH=$JAVA_HOME/bin:$PATH  \
    CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
WORKDIR /home/springboot/                                                      # Set the working directory of the image.
```

2. Press **Esc** and enter **:wq** to save the Dockerfile and exit.
3. Build an image.

   docker build -t openjdk:8 .
4. Run **docker images** to check whether the image is built successfully.

   Figure 2-1 Checking the built image



Step 5 Log in to the SWR console and create an organization.

Step 6 Push the image to the organization created in **Step 5**.

1. Log in to the SWR console as **root**.
2. Tag the image.
3. Push the image to the organization created in **Step 5**.

After an image is pushed, you can find it on the **My Images** page of the SWR console.

Step 7 (Optional) Use the pushed image to **deploy a workload in CCE**.

----End

3 Creating Multi-Architecture Images

Scenarios

When images are required in the x86 and Arm architectures, and different image tags need to be set, you can create multi-architecture images and manage these images of the same tag using the **docker manifest** command.

Preparations

1. Check the Docker version.

   Ensure that you use Docker 19.03 or later that supports the manifest extension. Run the following command to view the version:

   ```
   docker --version
   ```

2. If your Docker version is earlier than 19.03, set the **experimental** environment variable.

   Open the Docker daemon configuration file **/etc/docker/daemon.json**.

   ```
   vim /etc/docker/daemon.json
   ```

   Change the value of **experimental** to **true**. If this field does not exist, add it.

   ```
   "experimental": true
   ```

   Save the file and restart Docker to apply the modification.

   ```
   sudo systemctl restart docker
   ```

3. Check whether experimental Docker CLI features are enabled.

   ```
   echo $DOCKER_CLI_EXPERIMENTAL
   ```

   - If **enabled** is returned, experimental Docker CLI features are enabled.
   - If **disabled** or no information is returned, experimental Docker CLI features are not enabled.

Procedure

Step 1 Build and push two images to the image repository by following the instructions in **Creating a JDK8 Base Image and Pushing It to SWR**.

```
docker push ${repository_url}/${organization}/${image_name}:${image_tag}-amd64
docker push ${repository_url}/${organization}/${image_name}:${image_tag}-arm64
```


Step 2 Build a manifest for the multi-architecture images.

1. Create a manifest.
```
docker manifest create --amend --insecure ${repository_url}/${organization}/${image_name}:${image_tag} ${repository_url}/${organization}/${image_name}:${image_tag}-amd64 ${repository_url}/${organization}/${image_name}:${image_tag}-arm64
```
2. Modify the manifest to add **arch** information.
```
docker manifest annotate ${repository_url}/${organization}/${image_name}:${image_tag} ${repository_url}/${organization}/${image_name}:${image_tag}-amd64 --arch amd64
docker manifest annotate ${repository_url}/${organization}/${image_name}:${image_tag} ${repository_url}/${organization}/${image_name}:${image_tag}-arm64 --arch arm64
```

Step 3 Push the manifest.
```
docker manifest push ${repository_url}/${organization}/${image_name}:${image_tag}
```

----End

Downloading an Image Across CPU Architectures

Use the **--platform** parameter to specify the architecture of the image to be pulled.

```
docker pull ${repository_url}/${organization}/${image_name}:${image_tag} --platform=linux/amd64
```

--platform parameter description:

- --platform=linux/amd64: Linux system of the x86_64 architecture
- **--platform=linux/arm64**: Linux system of the ARM64 architecture

4 Configuring Access Network

4.1 Overview

For SWR Basic Edition, if your container engine client is installed on a CCE node or an ECS, you have three choices for the network used to pull and push images.

- **Private Network Access**
- **Public Network Access**
- **VPN/Direct Connect Access**

4.2 Private Network Access

SWR supports two ways of private network access.

- **Default way**
- **Access through VPC Endpoint**

Access Through the Default Way

If your container engine client is installed on a CCE node or an ECS that is in the same region as the image repository, you can push or pull images over a private network.

Private network access does not need any configurations.

CAUTION

If you access OBS through fixed VPC endpoints, you need to configure policies to allow access to OBS bucket clusters that store SWR container images. Otherwise, the images may fail to be pulled. You can **submit a service ticket** to obtain information about SWR's OBS bucket clusters in different regions.

Access Through VPC Endpoint

Scenarios

VPC Endpoint is used to establish a convenient, secure, and private connection channel between the VPC and SWR. If you have higher security requirements, you can purchase a VPC endpoint service to access SWR.

Procedure

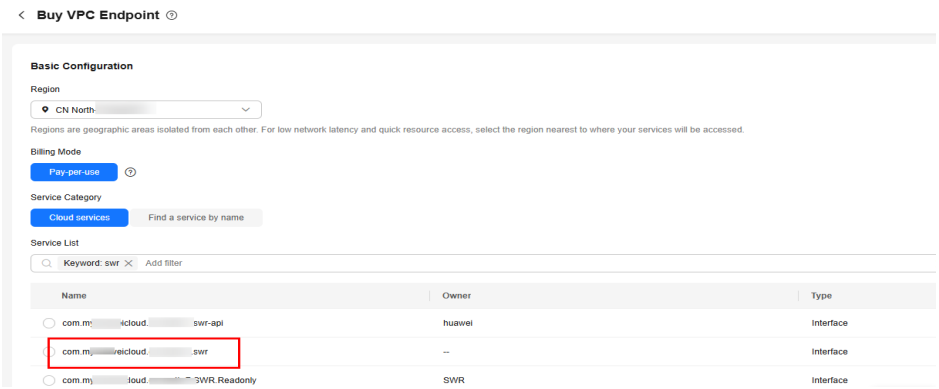
Step 1 Log in to the **management console**.

Step 2 Click  in the upper left corner and select the desired region and project.

Step 3 In the **Service List** in the upper left corner, choose **Networking** > **VPC Endpoint**.

Step 4 On the **VPC Endpoints** page, click **Buy VPC Endpoint**. In the **Select List**, select the VPC endpoint service below and then configure other parameters as needed:

com.myhuaweicloud.{region_id}.swr



NOTE

In some regions, the VPC endpoint service for SWR is not displayed in the **Service List**. You can search for it by name.

Table 4-1 VPC endpoint services for SWR in each region

Region Name	VPC Endpoint Service for SWR
CN Southwest-Guiyang1	com.myhuaweicloud.cn-southwest-2.swr
CN East2	com.myhuaweicloud.cn-east-4.swr
CN South-Guangzhou	com.myhuaweicloud.cn-south-1.swr
ME-Riyadh	com.myhuaweicloud.me-east-1.swr
AF-Johannesburg	com.myhuaweicloud.af-south-1.swr
LA-Mexico City2	com.myhuaweicloud.la-north-2.swr

Region Name	VPC Endpoint Service for SWR
AP-Singapore	com.myhuaweicloud.ap-southeast-3.swr
CN East-Shanghai1	com.myhuaweicloud.cn-east-3.swr
CN North-Beijing4	com.myhuaweicloud.cn-north-4.swr
CN North-Ulanqab1	com.myhuaweicloud.cn-north-9.swr
CN North-Guangzhou-InvitationOnly	com.myhuaweicloud.cn-south-4.swr
CN North3	com.myhuaweicloud.cn-north-12.swr
LA-Sao Paulo1	com.myhuaweicloud.sa-brazil-1.swr
Automotive II	com.myhuaweicloud.cn-southwest-3.swr
Qingdao	com.myhuaweicloud.cn-east-5.swr
CN-Hong Kong	com.myhuaweicloud.ap-southeast-1.swr
AP-Bangkok	com.myhuaweicloud.ap-southeast-2.swr
AP-Manila	com.myhuaweicloud.ap-southeast-5.swr
AF-Cairo	com.myhuaweicloud.af-north-1.swr
TR-Istanbul	com.myhuaweicloud.tr-west-1.swr
AP-Jakarta	com.myhuaweicloud.ap-southeast-4.swr
LA-Santiago	com.myhuaweicloud.la-south-2.swr

Step 5 Click **Next**.

Step 6 Confirm the order details and click **Submit**.

----End

4.3 Public Network Access

Scenarios

If your container engine client is installed on a CCE node or an ECS that is in a different region from the image repository, you can push or pull images over a public network. An EIP needs to be bound to the CCE node or ECS. For public network access, there are two scenarios.

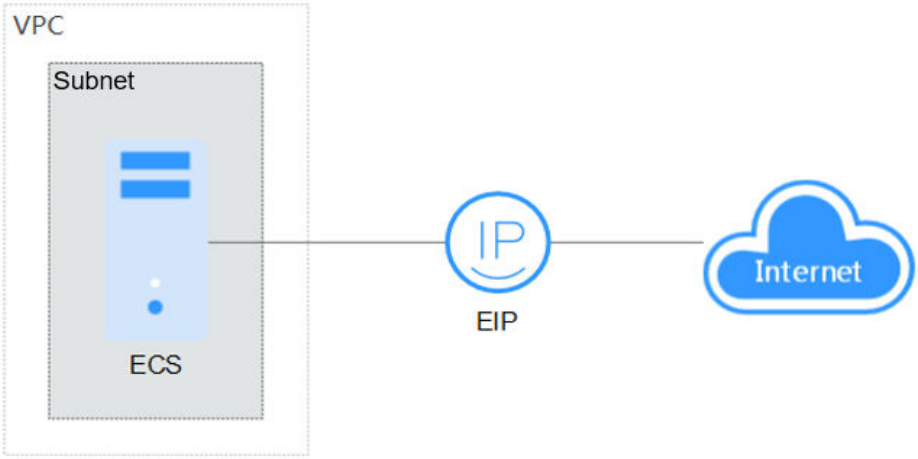
- **Public Network Access for Single ECS**

- Public Network Access for Multiple ECSs

Public Network Access for Single ECS

If an ECS needs to access a public network, you can bind it to an EIP. Huawei Cloud provides multiple billing modes (such as pay-per-use and pay-per-traffic). You can select one as required and flexibly unbind an unnecessary EIP.

Figure 4-1 Network topology



Step 1 Log in to the management console.

Step 2 Click in the upper left corner and select the desired region and project.

Step 3 Click and choose **Computing** > **Elastic Cloud Server**.

Step 4 In the ECS list, select the ECS to which an EIP is to be bound, and choose **More** > **Manage Network** > **Bind EIP** in the **Operation** column.

Step 5 Select an EIP and click **OK**.

Step 6 After the ECS is bound to the EIP, you can view the bound EIP in the ECS list.

NOTE

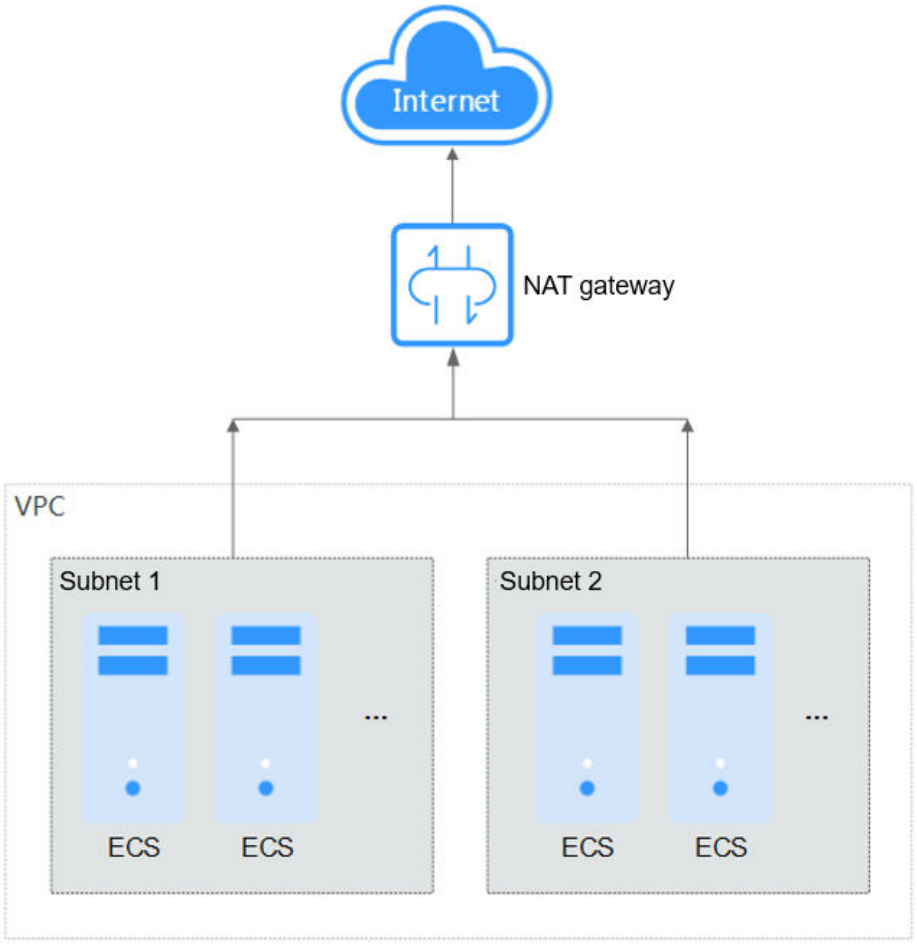
If no EIP is available in the current region, the EIP list is empty. In this case, purchase an EIP and bind it again.

----End

Public Network Access for Multiple ECSs

If all ECSs in your VPC need to access a public network, you can use NAT Gateway and configure SNAT rules by subnet to easily build a public network egress for the VPC. If no SNAT rule is configured, external users cannot directly access the public network IP address of the NAT gateway through a public network, which makes ECS more secure compared with public network access through an EIP.

Figure 4-2 Network topology



Step 1 Bind the ECS to an EIP. For details, see **Public Network Access for Single ECS**.

Step 2 Create a NAT gateway.

1. Log in to the management console.
2. Click in the upper left corner and select the desired region and project.
3. Click in the upper left corner and choose **Networking** > **NAT Gateway** in the service list.
4. On the displayed page, click **Buy Public NAT Gateway**.
5. Configure the parameters as prompted.

Step 3 Configure SNAT rules and bind EIPs to subnets.

1. Log in to the management console.
2. Click in the upper left corner and select the desired region and project.
3. Click in the upper left corner and choose **Networking** > **NAT Gateway** in the service list.
4. On the displayed page, click the name of the NAT gateway for which you want to add the SNAT rule.

5. On the **SNAT Rules** tab, click **Add SNAT Rule**.
6. Configure the parameters as prompted.

----End

CAUTION

If you access OBS by configuring the local **/etc/hosts** file, add the domain names of the OBS bucket clusters that store container images to this file. Otherwise, the images may fail to be pulled. You can **submit a service ticket** to obtain information about SWR's OBS bucket clusters in different regions.

4.4 VPN/Direct Connect Access

Scenarios

If your local data center or private network cannot access SWR through a public network, you can use Direct Connect or VPN to connect to Huawei Cloud VPC and use a VPC endpoint to access SWR.

This applies only to pushing images. To pull images, you also need to .

Procedure

Step 1 **Create a VPC.** For details, see **Creating a VPC**.

Step 2 **Create a Direct Connect connection or VPN so that the data center can connect to the VPC through Direct Connect or VPN.**

Step 3 **Buy a VPC endpoint.**

1. Log in to the management console.
2. Click  in the upper left corner and select the desired region and project.
3. In the **Service List** in the upper left corner, choose **Networking** > **VPC Endpoint**.
4. On the displayed page, click **Buy VPC Endpoint**.
5. Configure the parameters as prompted.
6. Click **Next**.
7. Confirm the order details and click **Submit**.

Step 4 **Obtain the private IP address and domain name for accessing the VPC.**

NOTE

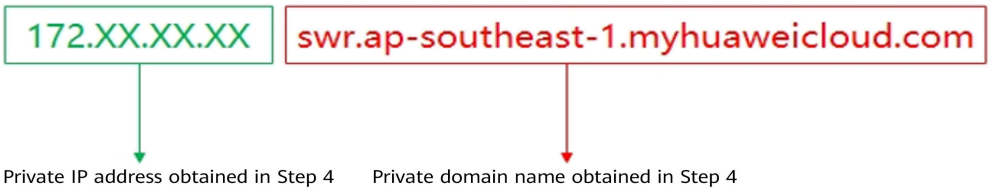
By default, the domain name resolution rule is automatically added to Huawei Cloud VPCs. You only need to configure hosts for non-Huawei Cloud endpoints.

1. Go to the VPC endpoint list.
2. Locate the purchased VPC endpoint and click the ID to go to the details page.

3. On the page displayed, you can view the IP address and private domain name of the VPC endpoint.

Step 5 **Configure hosts of the local data center.** The hosts IP address consists of the IP address and private domain name of the VPC endpoint. Example:

Figure 4-3 Example hosts



CAUTION

In this section, **172.xx.xx.xx swr.ap-southeast-1.myhuaweicloud.com** is only an example. Replace it with the actual IP address and private domain name.

There are two configuration methods:

Configuring Hosts for Linux

Customizing DNS Hosts

- **Configuring Hosts for Linux:**

1. Run the following command to open the **/etc/hosts** file:
```
sudo vim /etc/hosts
```
2. Add a custom domain name in the format of *xx.xx.xx.xx* **swr.***xx* **-***xx***.myhuaweicloud.com**.

   NOTE

   xx.xx.xx.xx and **swr.***xx* **-***xx***.myhuaweicloud.com** indicate the IP address and domain name obtained in **Step 4**, respectively.
3. Run the following command to restart the network.

```
sudo/etc/init.d/networking restart
```

- **Customizing DNS Hosts:**

1. Obtain the IP address of the VPC endpoint by referring to **4**.
2. Configure DNS forwarding rules on the DNS server in the local data center.

   The method of configuring DNS forwarding rules varies depending onOSs. For details, see the operation guide of the corresponding DNS software.

   This step uses the Linux OS and Bind (common DNS software) as an example.

   a. Edit the **/etc/named.conf** file to add a zone.
```
zone " swr.xx-xx.myhuaweicloud.com " IN {
  type master;
  file " /var/named/swr.xx-xx.myhuaweicloud.com.zone";
};
```


NOTE

swr.*xx-xx*.myhuaweicloud.com indicates the private domain name obtained in **Step 4**.

b. Configure forward DNS resolution. Create a file **/var/named/swr.*xx-xx*.myhuaweicloud.com.zone** mentioned in **a**.

```
$TTL    604800
@ IN     SOA       swr.xx-xx.myhuaweicloud.com. root.localhost. (
                      2         ; Serial
                 604800         ; Refresh
                  86400         ; Retry
                2419200         ; Expire
                 604800 )       ; Negative Cache TTL
;
@    IN   NS   swr.xx-xx.myhuaweicloud.com.
swr.xx-xx.myhuaweicloud.com.    IN   A     xx.xx.xx.xx
```

c. Restart the service.

/sbin/service named restart

NOTE

- You can query SWR endpoints in different regions in "Regions and Endpoints".
- If no DNS server is available in the local data center, add the endpoint IP address for accessing DNS to the **/etc/resolv.conf** file of the local data center.
- **swr.*xx-xx*.myhuaweicloud.com** indicates the IP address obtained in **Step 4**.

Step 6 Run the following command to verify the configuration and check the output.

ping swr.xx -xx.myhuaweicloud.com

Step 7 Use this domain name (**swr.xx -xx.myhuaweicloud.com**) in the later access to SWR.

----End

5 Migrating Container Images

5.1 Overview

Challenges

Containers are growing in popularity. Many enterprises choose to build their own Kubernetes clusters. However, the O&M workload of on-premises clusters is heavy, and O&M personnel need to configure the management systems and monitoring solutions by themselves. For enterprises, managing a large number of images requires high O&M, labor, and management costs, and the efficiency is low.

SoftWare Repository for Container (SWR) manages container images that function on multiple architectures, such as Linux and Arm. Enterprises can migrate their image repositories to SWR to reduce costs.

This section describes four ways to migrate image repositories to SWR smoothly. You can select one as needed.

Migration Solutions

Table 5-1 Comparison of migration solutions and application scenarios

Solution	Application Scenario	Precautions
Migrating images to SWR using Docker commands	Small quantity of images	• Disk storage leads to the timely deletion of local images and time-cost flushing. • Docker daemon strictly restricts the number of concurrent pull/push operations, so high-concurrency synchronization cannot be performed. • Scripts are complex because HTTP APIs are needed to perform the operations that cannot be implemented through Docker CLI.

Solution	Application Scenario	Precautions
Migrating images to SWR using image-syncer	A large number of images	• Images can be synchronized from multiple source repositories to multiple destination repositories. • Docker Registry V2-based image repositories (such as Docker Hub, Quay, and Harbor) can be migrated to SWR. • Memory- and network-dependent synchronization is fast. • A file records the blob information about synchronized images to avoid repeated synchronization. • You can modify the number of concurrent synchronization tasks in a configuration file. • Automatic retry of failed synchronization tasks can resolve most image synchronization issues caused by network jitters. • Docker or other programs are not required.

5.2 Migrating Images to SWR Using Docker Commands

Scenarios

SWR provides easy-to-use image hosting and efficient distribution services. If small quantity of images need to be migrated, enterprises can use the **docker pull/push** command to migrate images to SWR.

Procedure

Step 1 Pull images from the source repository.

Run the **docker pull** command to pull the images.

Example: **docker pull nginx:latest**

Run the **docker images** command to check whether the images are successfully pulled.

```
# docker images
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
nginx                         latest    22f2bf2e2b4f   5 hours ago     22.8MB
```

Step 2 Push the images pulled in **Step 1** to SWR.

1. Log in to the VM where the destination container is located and log in to SWR.
2. Tag the images.

   docker tag [Image name:Tag name] [Image repository address]/[Organization name]/[Image name:Tag name]

   Example:
3. Run the following command to push the images to the destination image repository.

   docker push [Image repository address]/[Organization name]/[Image name:Tag name]

   Example:
4. Check whether the following information is returned. If yes, the push is successful.

   ```
   fbce26647e70: Pushed
   fb04ab8effa8: Pushed
   8f736d52032f: Pushed
   009f1d338b57: Pushed
   678bbd796838: Pushed
   d1279c519351: Pushed
   f68ef921efae: Pushed
   v1: digest: sha256:0cdfc7910db531bfa7726de4c19ec556bc9190aad9bd3de93787e8bce3385f8d size: 1780
   ```

   To view the pushed image, refresh the **My Images** page.

----End

5.3 Migrating Images Using image-syncer

Scenarios

If a small quantity of images need to be migrated, you can use Docker commands. However, for thousands of images and several TBs of image repository data, it takes a long time and even data may be lost. In this case, you can use the open source image migration tool **image-syncer**.

Procedure

Step 1 **Download image-syncer**. Then, decompress and run it.

The following uses image-syncer v1.5.0 as an example.

wget https://github.com/AliyunContainerService/image-syncer/releases/download/v1.5.0/image-syncer-v1.5.0-linux-amd64.tar.gz

tar -zvxf image-syncer-v1.5.0-linux-amd64.tar.gz

Step 2 Create **auth.json**, the authentication information file of the image registries.

image-syncer can migrate Docker Registry V2-based repositories. Write the authentication information of the source and destination registries. The following is an example:

```
{
```

Step 3 Create **images.json**, the image synchronization description file.

In the following example, the source registry address is on the left, and the destination registry address is on the right. image-syncer also supports other description modes. For details, see **README.md**.

```
{
 }
```

Step 4 Run the following command to migrate the images to SWR:

Table 5-2 Command parameter description

Parameter	Description
--config	Path of the configuration file. This file needs to be created before you start the synchronization. By default, the configuration file is the **config.json** file in the current directory. (This parameter can be replaced with parameters **--auth** and **--images** which represent authentication information and repository synchronization rules respectively.)
--images	Path of the image rules file. This file needs to be created before you start the synchronization. By default, the rule file is the **images.json** file in the current directory.
--auth	Path of the authentication file. This file needs to be created before you start the synchronization. By default, the authentication file is the **auth.json** file in the current directory.
--log	Path of the log file. Logs will be printed to Stderr by default. You need to use **cat** to check logs.
--namespace	default-namespace. default-namespace can also be set by environment variable **DEFAULT_NAMESPACE**. If both the parameter and environment variable are set, the parameter will take precedence. default-namespace will work only if the destination registry is empty but the default-registry is not empty.
--proc	Number of goroutines. The default value is **5**. You are advised to use this value and do not change it.
--retries	Number of retries. The default value is **2**. The retries of failed sync tasks will start after all sync tasks are executed once. Retrying sync tasks will resolve most occasional network problems during synchronization.

Parameter	Description
--registry	default-registry. default-registry can also be set by environment variable **DEFAULT_REGISTRY**. If both the parameter and environment variable are set, the parameter will take precedence. default-registry will work only if the destination registry is empty but the default-namespace is not empty.

Step 5 Log in to the destination image registry to check the migrated images.

----End

6 Automatically Adding Image Retention Policies Using Cloud Custodian

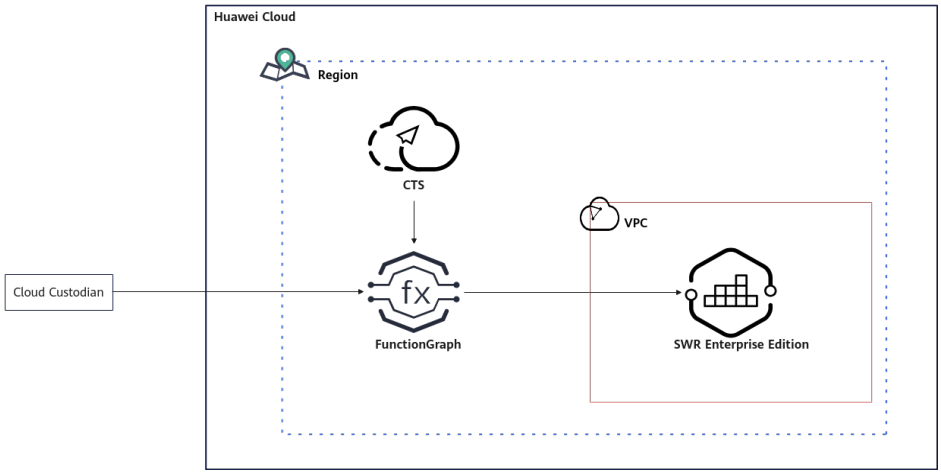
NOTE

This section applies only to SWR Enterprise Edition.

Many enterprises use cloud services provided by different cloud vendors. In such a hybrid cloud environment, alongside the security approaches provided by each cloud vendor, enterprises need strict governance over the cloud infrastructure. SWR is an important container service, and its security is critical. It is necessary to monitor SWR to prevent any access and permission vulnerabilities.

Cloud Custodian provides an open-source rule engine that can automatically check and govern cloud resources based on predefined security policies and compliance requirements. It can be used to control access to and govern resources in SWR. Cloud Custodian allows you to set rules to verify the environment based on the defined security and compliance standards. It helps follow security rules, manage tags, recycle unused resources, and control costs. It also provides an interface for you to easily implement consistent security policies and operational specifications in hybrid cloud environments.

Figure 6-1 Architecture of how Cloud Custodian is used to manage SWR resources



In this figure, there are the following cloud services:

- CTS records operations on cloud resources in your account. You can use the logs to perform security analysis, track resource changes, audit compliance, and locate faults.
- FunctionGraph hosts and computes event-driven functions in a serverless context while ensuring high availability, high scalability, and zero maintenance. All you need to do is write your code and set conditions.
- SWR Enterprise Edition provides secure and dedicated hosting services. You can host cloud native artifacts that comply with the OCI standard, such as container images and Helm charts.

Procedure

Step 1 Install Python. This is the running environment Cloud Custodian depends on.

NOTE

You are advised to use Python 3.11 and develop the Python application in a virtual environment.

For other Python versions, install them according to the official Python documentation. If you use native Python 3.11, run the commands below to create and activate a virtual environment.

```
# Create a virtual environment.
python -m venv custodian
# Activate a virtual environment (Linux).
source custodian/bin/activate
# Activate a virtual environment (Windows).
custodian\Scripts\activate.bat
If you use Conda or Miniconda, you can also run the commands below to create and activate a virtual environment.
# Create a virtual environment.
conda create -n custodian python=3.11
# Activate a virtual environment.
conda activate custodian
```

Step 2 Install Cloud Custodian.

1. Download Cloud Custodian.
   ```
   git clone https://github.com/huaweicloud/cloud-custodian.git
   cd cloud-custodian
   ```
2. Install Python dependencies.
   ```
   pip install -e .
   pip install -e tools/c7n_huaweicloud/.
   ```

For more details, see **Cloud Custodian official documentation**.

Step 3 Check whether Cloud Custodian is installed.

```
custodian schema huaweicloud.swr-ee
```

If the following information is displayed, the installation is successful.

```
$ custodian schema huaweicloud.swr-ee

Help
----

Huawei Cloud SWR Enterprise Edition Resource Manager.

This class manages SWR Enterprise Edition repositories on HuaweiCloud.
It provides functionality for discovering, filtering, and managing SWR repositories.

huaweicloud.swr-ee:
  actions:
  - notify-message
  - notify-message-structure
  - notify-message-template
  - webhook
  filters:
  - age
  - event
  - exempted
  - list-item
  - reduce
  - restricted
  - value

(custodian)
```

Step 4 Define a Cloud Custodian policy.

```
policies:
  - name: swr-ee-event
    resource: huaweicloud.swr-ee-namespace
    mode:
      type: cloudtrace
      xrole: fgs_default_agency
      events:
        - source: 'SWR.namespace'
          event: 'createNamespace'
          ids: 'resource_name'
    actions:
      - type: set-lifecycle
        rules:
          - template: latestPushedK
            params:
              latestPushedK: 50
            scope_selectors:
              repository:
                - kind: doublestar
                  pattern: '**'
            tag_selectors:
              - kind: doublestar
                decoration: matches
                pattern: '**'
```

NOTE

When you create a namespace of SWR Enterprise Edition, the policy automatically sets a retention policy for the namespace to retain the latest 50 artifact tags.

The following are two example policies for your reference:

Policy 1: Retain images in images repositories that have been created for more than 90 days.

```
policies:
  - name: swr-ee-repos
    resource: huaweicloud.swr-ee
    filters:
      - type: age
        days: 90
        op: gt
```

Policy 2: Retain images whose artifact tags match **release***.

```
policies:
  - name: swr-ee-set-immutability-rules
    resource: huaweicloud.swr-ee-namespace
```



```
actions:
  - type: set-immutability
    state: True
    scope_selectors:
      repository:
        - kind: doublestar
          pattern: '**'
    tag_selectors:
      - kind: doublestar
        decoration: matches
        pattern: '{release*}'
```

For more policy configurations, see **Cloud Custodian official documentation**.

Step 5 Run the custodian command to enable the policy.

1. Before executing the policy, configure environment variables in the terminal. The commands below are used in Linux. In Windows, replace **export** with **set**.

```
# Configure the AK/SK of the Huawei Cloud account.
export HUAWEI_ACCESS_KEY_ID={your-ak}
export HUAWEI_SECRET_ACCESS_KEY={your-sk}
# Configure the default region where resources are located, for example, ap-southeast-1. You can obtain the region from the web page URL.
export HUAWEI_DEFAULT_REGION={your-region}
```

2. Enable the policy.

```
custodian run --output-dir=<output_directory> <policy_name>.yaml
```

----End