Cloud Container Engine

Best Practices

Issue	01
Date	2024-05-31

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property 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 Cloud 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.

Contents

1 Checklist for Deploying Containerized Applications in the Cloud

Overview

Security, efficiency, stability, and availability are common requirements on all cloud services. To meet these requirements, the system availability, data reliability, and O&M stability must be coordinated. This checklist describes the check items for deploying containerized applications on the cloud to help you efficiently migrate services to CCE, reducing potential cluster or application exceptions caused by improper use.

Check Items

Table 1-1 System availability

Category	Check Item	Type	Impact	FAQ & Example
Cluster	Before creating a cluster, properly plan the node network and container network based on service requirements to allow subsequent service expansion.	Network planning	If the subnet or container CIDR block where the cluster resides is small, the number of available nodes supported by the cluster may be less than required.	• **Network Planning** • **Planning CIDR Blocks for a Cluster** • **How Do I Set the VPC CIDR Block and Subnet CIDR Block for a CCE Cluster?**

Category	Check Item	Type	Impact	FAQ & Example
	Before creating a cluster, properly plan CIDR blocks for the related Direct Connect, peering connection, container network, service network, and subnet to avoid IP address conflicts.	Network planning	If CIDR blocks are not properly set and IP address conflicts occur, service access will be affected.	• **Connectivity** • **Planning CIDR Blocks for a Cluster**
	When a cluster is created, the default security group is automatically created and bound to the cluster. You can set custom security group rules based on service requirements.	Deployment	Security groups are key to security isolation. Improper security policy configuration may cause security risks and service connectivity problems.	• **Security Groups and Security Group Rules** • **How Do I Prevent Cluster Nodes from Being Exposed to Public Networks?**

Category	Check Item	Type	Impact	FAQ & Example
	Enable the multi-master node mode, and set the number of master nodes to **3** when creating a cluster.	Reliability	After the multi-master node mode is enabled, three master nodes will be created. If a master node is faulty, the cluster can still be available without affecting service functions. In commercial scenarios, it is advised to enable the multi-master node mode.	**How Do I Check Whether a Cluster Is in Multi-Master Mode?** Once a cluster is created, the number of master nodes cannot be changed. Exercise caution when setting the number of master nodes.
	When creating a cluster, select a proper network model as needed. • Select **VPC network** or **Tunnel network** for your CCE standard cluster. • Select **Cloud Native Network 2.0** for your CCE Turbo cluster.	Deployment	After a cluster is created, the network model cannot be changed. Exercise caution when selecting a network model.	**Network Model Comparison**

Category	Check Item	Type	Impact	FAQ & Example
Workload	When creating a workload, set the CPU and memory limits to improve service robustness.	Deployment	When multiple applications are deployed on the same node, if the upper and lower resource limits are not set for an application, resource leakage occurs. As a result, resources cannot be allocated to other applications, and the application monitoring information will be inaccurate.	None
	When creating a workload, you can set probes for container health check, including **liveness probe** and **readiness probe**.	Reliability	If the health check function is not configured, a pod cannot detect service exceptions or automatically restart the service to restore it. This results in a situation where the pod status is normal but the service in the pod is abnormal.	• **Setting Health Check for a Container** • **Enabling ICMP Security Group Rules**

Category	Check Item	Type	Impact	FAQ & Example
	When creating a workload, select a proper access mode (Service). Currently, the following types of Services are supported: ClusterIP, NodePort, DNAT, and LoadBalancer.	Deployment	Improper Service configuration may cause logic confusion for internal and external access and resource waste.	• **Network Overview**
	When creating a workload, do not set the number of replicas for a single pod. Set a proper node scheduling policy based on your service requirements.	Reliability	For example, if the number of replicas of a single pod is set, the service will be abnormal when the node or pod is abnormal. To ensure that your pods can be successfully scheduled, ensure that the node has idle resources for container scheduling after you set the scheduling rule.	None

Category	Check Item	Type	Impact	FAQ & Example
	Properly set affinity and anti-affinity.	Reliability	If affinity and anti-affinity are both configured for an application that provides Services externally, Services may fail to be accessed after the application is upgraded or restarted.	**Scheduling Policy (Affinity/Anti-affinity)** **Negative example:** For application A, nodes 1 and 2 are set as affinity nodes, and nodes 3 and 4 are set as anti-affinity nodes. Application A exposes a Service through the ELB, and the ELB listens to node 1 and node 2. When application A is upgraded, it may be scheduled to a node other than nodes 1, 2, 3, and 4, and it cannot be accessed through the Service. **Cause:** Scheduling of application A does not need to meet both affinity and anti-affinity policies. A node will be selected for application A according to either of the policies. In this example, the node selection is based on the anti-affinity scheduling policy.

Category	Check Item	Type	Impact	FAQ & Example
	When creating a workload, set the pre-stop processing command (**Lifecycle** > **Pre-Stop**) to ensure that the services running in the pods can be completed in advance in the case of application upgrade or pod deletion.	Reliability	If the pre-stop processing command is not configured, the pod will be directly killed and services will be interrupted during application upgrade.	• **Setting Container Lifecycle Parameters** • **When Is Pre-stop Processing Used?**

Table 1-2 Data reliability

Category	Check Item	Type	Impact	FAQ & Example
Container data persistency	Select a proper data volume type based on service requirements.	Reliability	When a node is faulty and cannot be recovered, data in the local disk cannot be recovered. Therefore, you are advised to use cloud storage volumes to ensure data reliability.	• **Storage Overview**
Backup	Back up application data.	Reliability	Data cannot be restored after being lost.	**What Are the Differences Among CCE Storage Classes in Terms of Persistent Storage and Multi-node Mounting?**

Table 1-3 O&M reliability

Category	Check Item	Type	Impact	FAQ & Example
Project	The quotas of ECS, VPC, subnet, EIP, and EVS resources must meet customer requirements.	Deployment	If the quota is insufficient, resources will fail to be created. Specifically, users who have configured auto scaling must have sufficient resource quotas.	• **Which Resource Quotas Should I Pay Attention To When Using CCE?** • **Notes and Constraints**
	You are not advised to modify kernel parameters, system configurations, cluster core component versions, security groups, and ELB-related parameters on cluster nodes, or install software that has not been verified.	Deployment	Exceptions may occur on CCE clusters or Kubernetes components on the node, making the node unavailable for application deployment.	For details, see **High-Risk Operations and Solutions**. **Negative example:** 1. The container network is interrupted after the node kernel is upgraded. 2. The container network is interrupted after an open-source Kubernetes network add-on is installed on a node. 3. The **/var/paas** or **/mnt/paas/kubernetes** directory is deleted from a node, which causes exceptions on the node.

Category	Check Item	Type	Impact	FAQ & Example
	Do not modify information about resources created by CCE, such as security groups and EVS disks. Resources created by CCE are labeled **cce**.	Deployment	CCE cluster functions may be abnormal.	**Negative example:** 1. On the ELB console, a user changes the name of the listener created by CCE. 2. On the VPC console, a user modifies the security group created by CCE. 3. On the EVS console, a user deletes or uninstalls data disks mounted to CCE cluster nodes. 4. On the IAM console, a user deletes **cce_admin_trust**. All the preceding actions will cause CCE cluster functions to be abnormal.

Category	Check Item	Type	Impact	FAQ & Example
Proactive O&M	CCE provides multi-dimensional monitoring and alarm reporting functions, allowing users to locate and rectify faults as soon as possible. • Application Operations Management (AOM): The default basic resource monitoring of CCE covers detailed container-related metrics and provides alarm reporting functions. • Open source Prometheus: A monitoring tool for cloud native applications. It integrates an independent alarm system to provide more	Monitoring	If the alarms are not configured, the standard of container cluster performance cannot be established. When an exception occurs, you cannot receive alarms and will need to manually locate the fault.	• **Monitoring Overview** • **Monitoring Custom Metrics Using the Cloud Native Monitoring Add-on**

Category	Check Item	Type	Impact	FAQ & Example
	flexible monitoring and alarm reporting functions.			

2 Containerization

2.1 Containerizing an Enterprise Application (ERP)

2.1.1 Solution Overview

This chapter provides CCE best practices to walk you through the application containerization.

What Is a Container?

A container is a lightweight high-performance resource isolation mechanism implemented based on the Linux kernel. It is a built-in capability of the operating system (OS) kernel.

CCE is an enterprise-class container service based on open-source Kubernetes. It is a high-performance and high-reliability service through which enterprises can manage containerized applications. CCE supports native Kubernetes applications and tools, allowing you to easily set up a container runtime in the cloud.

Why Is a Container Preferred?

- More efficient use of system resources

  A container does not require extra costs such as fees for hardware virtualization and those for running a complete OS. Therefore, a container has higher resource usage. Compared with a VM with the same configurations, a container can run more applications.

- Faster startup

  A container directly runs on the host kernel and does not need to start a complete OS. Therefore, a container can be started within seconds or even milliseconds, greatly saving the development, testing, and deployment time.

- Consistent runtime environment

  A container image provides a complete runtime environment to ensure environment consistency. In this case, problems (for example, some code runs properly on machine A but fails to run on machine B) will not occur.

- Easier application migration, maintenance, and scaling

  A consistent runtime environment makes application migration easier. In addition, the in-use storage and image technologies facilitate the reuse of repeated applications and simplifies the expansion of images based on base images.

Containerization Modes

The following modes are available for containerizing applications:

- Mode 1: Containerize a single application as a whole. Application code and architecture remain unchanged.
- Mode 2: Separate the components that are frequently upgraded or have high requirements on auto scaling from an application, and then containerize these components.
- Mode 3: Transform an application to microservices and then containerize the microservices one by one.

Table 2-1 lists the advantages and disadvantages of the three modes.

Table 2-1 Containerization modes

Containerization Mode	Advantage	Disadvantage
Method 1: Containerize a single application as a whole.	• Zero modification on services: The application architecture and code require no change. • The deployment and upgrade efficiency is improved. Applications can be packed as container images to ensure application environment consistency and improve deployment efficiency. • Reduce resource costs: Containers use system resources more efficiently. Compared with a VM with the same configurations, a container can run more applications.	• Difficult to expand the entire architecture of an application. As the code size increases, code update and maintenance would be complicated. • Difficult to launch new functions, languages, frameworks, and technologies.

Containerization Mode	Advantage	Disadvantage
Method 2: Containerize first the application components that are frequently updated or have high requirements on auto scaling.	• Progressive transformation: Reconstructing the entire architecture involves a heavy workload. This mode containerizes only a part of components, which is easy to accept for customers. • Flexible scaling: Application components that have high requirements on auto scaling are containerized. When the application needs to be scaled, you only need to scale the containers, which is flexible and reduces the required system resources. • Faster rollout of new features: Application components that are frequently upgraded are containerized. In subsequent upgrades, only these containers need to be upgraded. This shortens the time to market (TTM) of new features.	Need to decouple some services.

Containerization Mode	Advantage	Disadvantage
Method 3: Transform an application to microservices and then containerize the microservices one by one.	• Independent scaling: After an application is split into microservices, you can independently increase or decrease the number of instances for each microservice. • Increased development speed: Microservices are decoupled from one another. Code development of a microservice does not affect other microservices. • Security assurance through isolation: For an overall application, if a security vulnerability exists, attackers can use this vulnerability to obtain the permission to all functions of the application. However, in a microservice architecture, if a service is attacked, attackers can only obtain the access permission to this service, but cannot intrude other services. • Breakdown isolation: If one microservice breaks down, other microservices can still run properly.	Need to transform the application to microservices, which involves a large number of changes.

Mode 1 is used as an example in this tutorial to illustrate how to containerize an enterprise resource planning (ERP) system.

2.1.2 Resource and Cost Planning

NOTICE

The fees listed here are estimates. The actual fees will be displayed on the Huawei Cloud console.

The required resources are as follows:

Table 2-2 Resource and cost planning

Resource	Description	Quantity	Estimated Fee (USD)
Elastic Cloud Server (ECS)	• Minimum ECS specifications: 4 vCPUs, 8 GB memory, Ubuntu 16.04 • EIP specification: billed by bandwidth, 5 Mbit/s • Pay-per-use recommended	1	0.124/hour
Cloud Container Engine (CCE)	• CCE cluster version: v1.21 • Minimum node specifications: 4 vCPUs, 8 GB memory, EulerOS 2.9 • Pay-per-use recommended	1	0.55/hour
Elastic Volume Service (EVS)	• EVS disk specification: 100 GB • Pay-per-use recommended	1	0.01/hour

2.1.3 Procedure

2.1.3.1 Containerizing an Entire Application

This tutorial describes how to containerize an ERP system by migrating it from a VM to CCE.

No recoding or re-architecting is required. You only need to pack the entire application into a container image and deploy the container image on CCE.

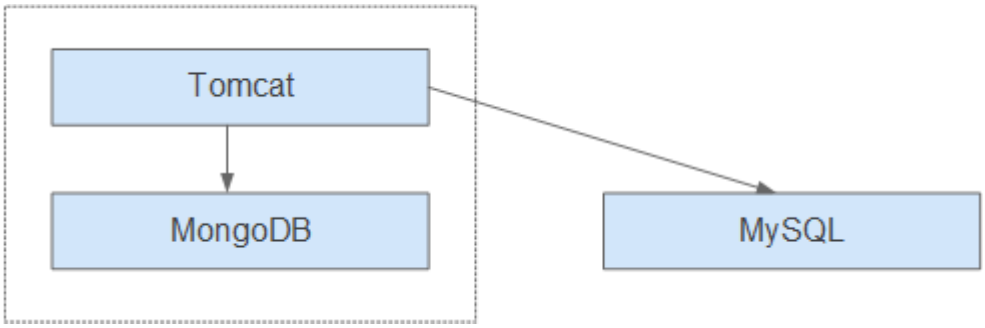
Introduction

In this example, the **enterprise management application** is developed by enterprise A. This application is provided for third-party enterprises for use, and enterprise A is responsible for application maintenance.

When a third-party enterprise needs to use this application, a suit of **Tomcat application** and **MongoDB database** must be deployed for the third-party

enterprise. The MySQL database, used to store data of third-party enterprises, is provided by enterprise A.

Figure 2-1 Application architecture



As shown in **Figure 2-1**, the application is a standard Tomcat application, and its backend interconnects with MongoDB and MySQL databases. For this type of applications, there is no need to split the architecture. The entire application is built as an image, and the MongoDB database is deployed in the same image as the Tomcat application. In this way, the application can be deployed or upgraded through the image.

- Interconnecting with the MongoDB database for storing user files.
- Interconnecting with the MySQL database for storing third-party enterprise data. The MySQL database is an external cloud database.

Benefits

In this example, the application was deployed on a VM. During application deployment and upgrade, a series of problems is encountered, but application containerization has solved these problems.

By using containers, you can easily pack application code, configurations, and dependencies and convert them into easy-to-use building blocks. This achieves the environmental consistency and version management, as well as improves the development and operation efficiency. Containers ensure quick, reliable, and consistent deployment of applications and prevent applications from being affected by deployment environment.

Table 2-3 Comparison between the two deployment modes

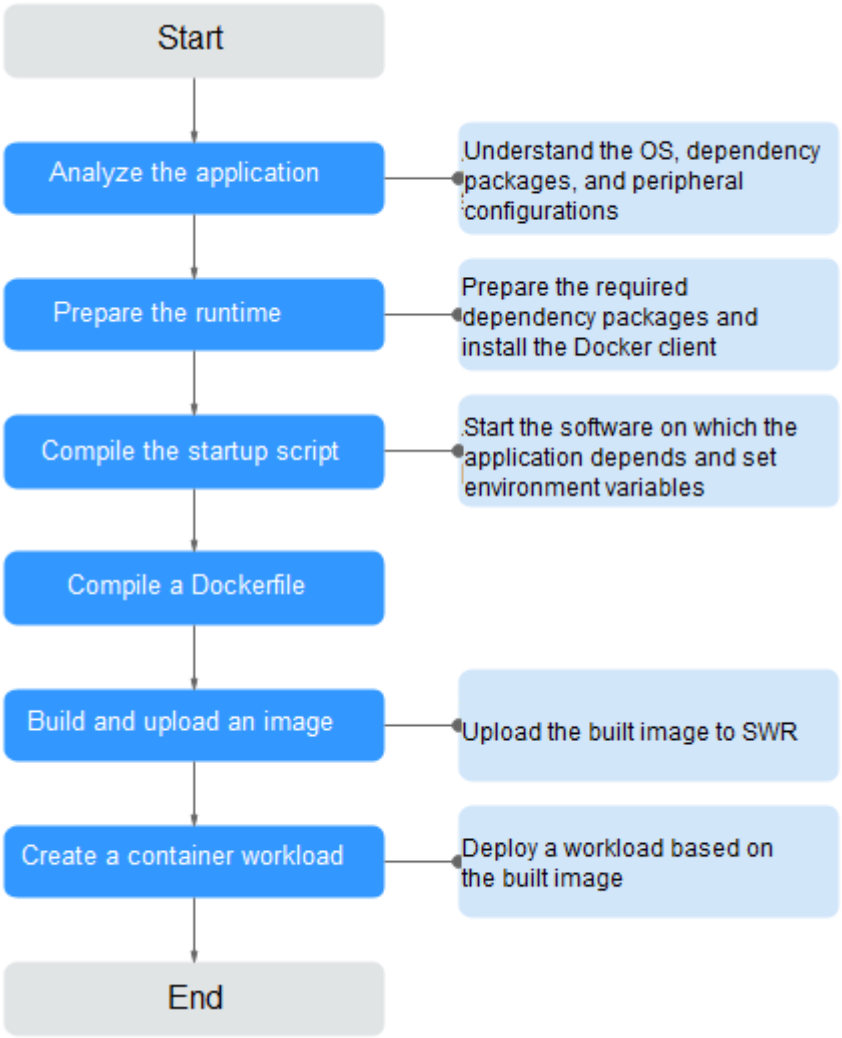
Category	Before: Application Deployment on VM	After: Application Deployment Using Containers
Deployment	High deployment cost. A VM is required for deploying a system for a customer.	More than 50% cost reduced. Container services achieve multi-tenant isolation, which allows you to deploy systems for different enterprises on the same VM.

Category	Before: Application Deployment on VM	After: Application Deployment Using Containers
Upgrade	Low upgrade efficiency. During version upgrades, log in to VMs one by one and manually configure the upgrades, which is inefficient and error-prone.	Per-second level upgrade. Version upgrades can be completed within seconds by replacing the image tag. In addition, CCE provides rolling updates, ensuring zero service downtime during upgrades.
Operation and maintenance (O&M)	High O&M cost. As the number of applications deployed for customer grows, the number of VMs that need to be maintained increases accordingly, which requires a large sum of maintenance cost.	Automatic O&M Enterprises can focus on service development without paying attention to VM maintenance.

2.1.3.2 Containerization Process

The following figure illustrates the process of containerizing an application.

Figure 2-2 Process of containerizing an application



2.1.3.3 Analyzing the Application

Before containerizing an application, analyze the running environment and dependencies of the application, and get familiar with the application deployment mode. For details, see **Table 2-4**.

Table 2-4 Application environment

Category	Sub-category	Description
Runtime environment	OS	OS that the application runs on, such as CentOS or Ubuntu. In this example, the application runs on CentOS 7.1.

Category	Sub-category	Description
	Runtime environment	The Java application requires Java Development Kit (JDK), the Go language requires GoLang, the web application requires Tomcat environment, and the corresponding version number needs to be confirmed. In this example, the web application of the Tomcat type is used. This application requires the runtime environment of Tomcat 7.0, and Tomcat requires JDK 1.8.
	Dependency package	Understand required dependency packages, such as OpenSSL and other system software, and their version numbers. In this example, no dependency package is required.
Deployment mode	Peripheral configurations	MongoDB database: In this example, the MongoDB database and Tomcat application are deployed on the same server. Therefore, their configurations can be fixed and there is no need to extract their configurations.

		External services with which the application needs to interconnect, such as databases and file systems.
		These configurations need to be manually configured each time you deploy an application on a VM. However, through containerized deployment, environment variables can be injected into a container, facilitating deployment.
		In this example, the application needs to interconnect with the MySQL database. Obtain the database configuration file. The server address, database name, database login username, and database login password are injected through environment variables.
		url=jdbc:mysql://*Server address*/*Database name* #Database connection URL
		username=**** #Username for logging in to the database
		password=**** #Password for logging in to the database

Category	Sub-category	Description
	Application configurations	Sort out the configuration parameters, such as configurations that need to be modified frequently and those remain unchanged during the running of the application. In this example, no application configurations need to be extracted. **NOTE** To avoid frequent image replacement, you are advised to classify configurations of the application. • For the configurations (such as peripheral interconnection information and log levels) that are frequently changed, you are advised to configure them as environment variables. • For the configurations that remain unchanged, directly write them into images.

2.1.3.4 Preparing the Application Runtime

After application analysis, you have gained the understanding of the OS and runtime required for running the application. Make the following preparations:

- **Installing Docker**: During application containerization, build a container image. To do so, you have to prepare a PC and install Docker on it.
- **Obtaining the base image tag**: Determine the base image based on the OS on which the application runs. In this example, the application runs on CentOS 7.1 and the base image can be obtained from an open-source image repository.
- **Obtaining the runtime**: Obtain the runtime of the application and the MongoDB database with which the application interconnects.

Installing Docker

Docker is compatible with almost all operating systems. Select a Docker version that best suits your needs.

NOTE

SWR uses Docker 1.11.2 or later to upload images.

You are advised to install Docker and build images as user **root**. Obtain the password of user **root** of the host where Docker is to be installed in advance.

Step 1 Log in as user **root** to the device on which Docker is about to be installed.

Step 2 Quickly install Docker on the device running Linux. You can also manually install Docker. For details, see **Docker Engine installation**.

curl -fsSL get.docker.com -o get-docker.sh

sh get-docker.sh

Step 3 Run the following command to query the Docker version:


```
docker version
Client:
Version: 17.12.0-ce
API Version:1.35
...
```

Version indicates the version number.

----End

Obtaining the Base Image Tag

Determine the base image based on the OS on which the application runs. In this example, the application runs on CentOS 7.1 and the base image can be obtained from an open-source image repository.

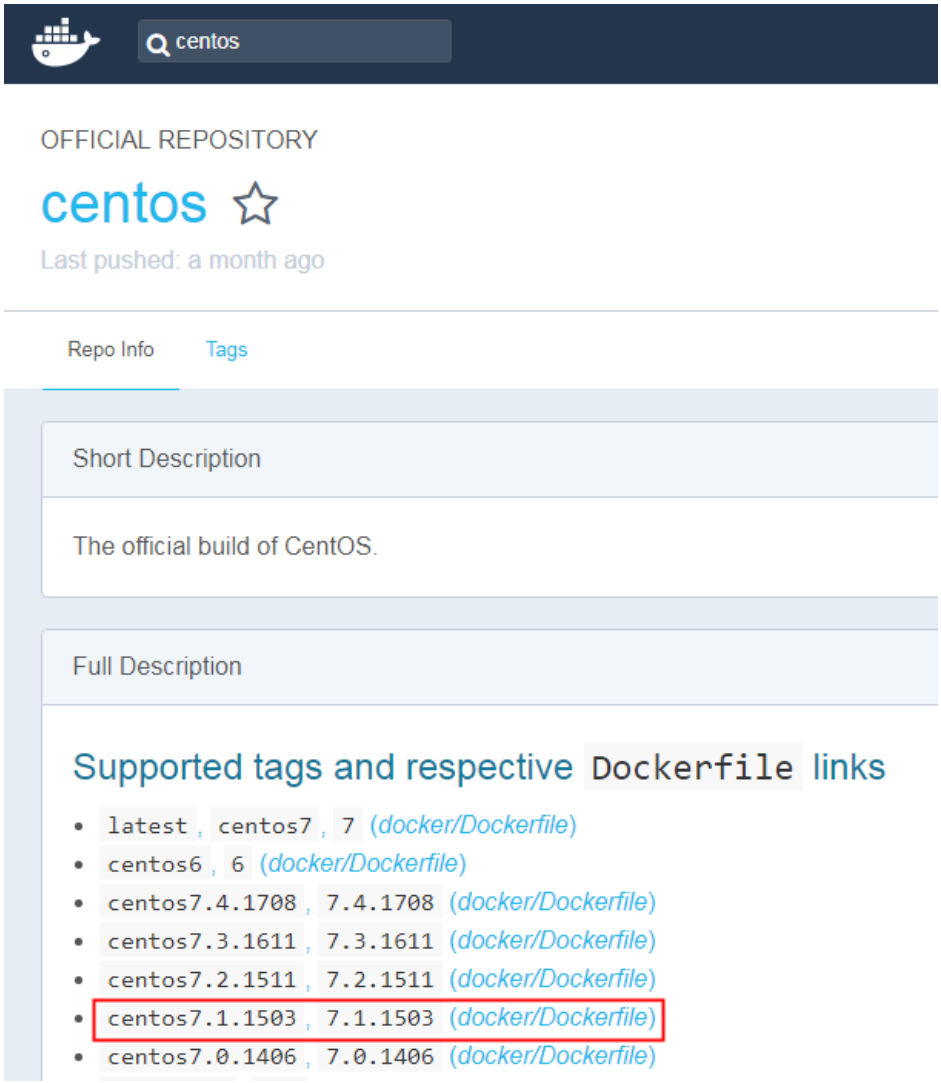
NOTE

Search for the image tag based on the OS on which the application runs.

Step 1 Visit the Docker website.

Step 2 Search for CentOS. The image corresponding to CentOS 7.1 is **centos7.1.1503**. Use this image name when editing the Dockerfile.

Figure 2-3 Obtaining the CentOS version



----End

Obtaining the Runtime

In this example, the web application of the Tomcat type is used. This application requires the runtime of Tomcat 7.0, and Tomcat requires JDK 1.8. In addition, the application must interconnect with the MongoDB database in advance.

NOTE

Download the environment required by the application.

Step 1 Download Tomcat, JDK, and MongoDB installation packages of the specific versions.

1. Download JDK 1.8.

   Download address: **https://www.oracle.com/java/technologies/jdk8-downloads.html**.

2. Download Tomcat 7.0 from **http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.82/bin/apache-tomcat-7.0.82.tar.gz**.

3. Download MongoDB 3.2 from **https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-3.2.9.tgz**.

Step 2 Log in as user **root** to the device running Docker.

Step 3 Run the following commands to create the directory where the application is to be stored: For example, set the directory to **apptest**.

mkdir apptest

cd apptest

Step 4 Use Xshell to save the downloaded dependency files to the **apptest** directory.

Step 5 Run the following commands to decompress the dependency files:

tar -zxf apache-tomcat-7.0.82.tar.gz

tar -zxf jdk-8u151-linux-x64.tar.gz

tar -zxf mongodb-linux-x86_64-rhel70-3.2.9.tgz

Step 6 Save the enterprise application (for example, **apptest.war**) in the **webapps/apptest** directory of the Tomcat runtime environment.

NOTE

apptest.war is used as an example only. Use your own application for actual configuration.

mkdir -p apache-tomcat-7.0.82/webapps/apptest

cp apptest.war apache-tomcat-7.0.82/webapps/apptest

cd apache-tomcat-7.0.82/webapps/apptest

./../../../jdk1.8.0_151/bin/jar -xf apptest.war

rm -rf apptest.war

----End

2.1.3.5 Compiling a Startup Script

During application containerization, prepare a startup script. The method of compiling this script is the same as that of compiling a shell script. The startup script is used to:

- Start up the software on which the application depends.
- Set the configurations that need to be changed as the environment variables.

NOTE

Startup scripts vary according to applications. Edit the script based on your service requirements.

Procedure

Step 1 Log in as user **root** to the device running Docker.

Step 2 Run the following commands to create the directory where the application is to be stored:

mkdir apptest

cd apptest

Step 3 Compile a script file. The name and content of the script file vary according to applications. Edit the script file based on your application. The following example is only for your reference.

vi start_tomcat_and_mongo.sh

```
#!/bin/bash
# Load system environment variables.
source  /etc/profile
# Start MongoDB. The data is stored in /usr/local/mongodb/data.
./usr/local/mongodb/bin/mongod --dbpath=/usr/local/mongodb/data --logpath=/usr/local/mongodb/logs --port=27017 –fork
# These three script commands indicate that the contents related to the MySQL database in the environment variables are written into the configuration file when Docker is started.
sed -i "s|mysql://.*/awcp_crmtile|mysql://$MYSQL_URL/$MYSQL_DB|g" /root/apache-tomcat-7.0.82/webapps/awcp/WEB-INF/classes/conf/jdbc.properties
sed -i "s|username=.*|username=$MYSQL_USER|g" /root/apache-tomcat-7.0.82/webapps/awcp/WEB-INF/classes/conf/jdbc.properties
sed -i "s|password=.*|password=$MYSQL_PASSWORD|g" /root/apache-tomcat-7.0.82/webapps/awcp/WEB-INF/classes/conf/jdbc.properties
# Start Tomcat.
bash /root/apache-tomcat-7.0.82/bin/catalina.sh run
```

----End

2.1.3.6 Compiling the Dockerfile

An image is the basis of a container. A container runs based on the content defined in the image. An image has multiple layers. Each layer includes the modifications made based on the previous layer.

Generally, Dockerfiles are used to customize images. Dockerfile is a text file and contains various instructions. Each instruction is used to build an image layer. That is, each instruction describes how to build an image layer.

This section describes how to compile a Dockerfile file.

NOTE

Dockerfiles vary according to applications. Dockerfiles need to be compiled based on actual service requirements.

For details on how to write a quality Dockerfile, see **Writing a Quality Dockerfile**.

Procedure

Step 1 Log in as the **root** user to the device running Docker.

Step 2 Compile a Dockerfile.

vi Dockerfile

The content is as follows:

```
# Centos:7.1.1503 is used as the base image.
FROM centos:7.1.1503
# Create a folder to store data and dependency files. You are advised to write multiple commands into one line to reduce the image size.
RUN mkdir -p /usr/local/mongodb/data \
 && mkdir -p /usr/local/mongodb/bin \
```



```
 && mkdir -p /root/apache-tomcat-7.0.82 \
 && mkdir -p /root/jdk1.8.0_151

# Copy the files in the apache-tomcat-7.0.82 directory to the container path.
COPY ./apache-tomcat-7.0.82 /root/apache-tomcat-7.0.82
# Copy the files in the jdk1.8.0_151 directory to the container path.
COPY ./jdk1.8.0_151 /root/jdk1.8.0_151
# Copy the files in the mongodb-linux-x86_64-rhel70-3.2.9 directory to the container path.
COPY ./mongodb-linux-x86_64-rhel70-3.2.9/bin /usr/local/mongodb/bin
# Copy start_tomcat_and_mongo.sh to the /root directory of the container.
COPY ./start_tomcat_and_mongo.sh /root/

# Enter Java environment variables.
RUN chown root:root -R /root \
 && echo "JAVA_HOME=/root/jdk1.8.0_151 " >> /etc/profile  \
 && echo "PATH=\$JAVA_HOME/bin:$PATH " >> /etc/profile  \
 && echo "CLASSPATH=.:\$JAVA_HOME/lib/dt.jar:\$JAVA_HOME/lib/tools.jar" >> /etc/profile  \
 && chmod +x /root \
 && chmod +x /root/start_tomcat_and_mongo.sh

# When the container is started, commands in start_tomcat_and_mongo.sh are automatically run. The file can be one or more commands, or a script.
ENTRYPOINT ["/root/start_tomcat_and_mongo.sh"]
```

In the preceding information:

- **FROM** statement: indicates that **centos:7.1.1503** is used as the base image.
- **Run** statement: indicates that a shell command is executed in the container.
- **COPY** statement: indicates that files in the local computer are copied to the container.
- **ENTRYPOINT** statement: indicates the commands that are run after the container is started.

----End

2.1.3.7 Building and Uploading an Image

This section describes how to build an entire application into a Docker image. After building an image, you can use the image to deploy and upgrade the application. This reduces manual configuration and improves efficiency.

NOTE

When building an image, ensure that files used to build the image are stored in the same directory.

Required Cloud Services

SoftWare Repository for Container (SWR) provides easy, secure, and reliable management over container images throughout their lifecycle, facilitating the deployment of containerized services.

Basic Concepts

- Image: A Docker image is a special file system that includes everything needed to run containers: programs, libraries, resources, settings, and so on. It also includes corresponding configuration parameters (such as anonymous volumes, environment variables, and users) required within a container runtime. An image does not contain any dynamic data, and its content remains unchanged after being built.

- Container: Images become containers at runtime, that is, containers are created from images. A container can be created, started, stopped, deleted, or suspended.

Procedure

Step 1 Log in as the **root** user to the device running Docker.

Step 2 Enter the **apptest** directory.

cd apptest

ll

Ensure that files used to build the image are stored in the same directory.

```
root@ecs-aos:~/apptest# ll
total 264456
drwxr-xr-x 5 root root      4096 Jan  2 19:59 ./
drwx------ 6 root root      4096 Jan  2 19:59 ../
drwxr-xr-x 9 root root      4096 Jan  2 19:55 apache-tomcat-7.0.82/
-rw-r--r-- 1 root root   8997403 Jan  2 19:52 apache-tomcat-7.0.82.tar.gz
-rw-r--r-- 1 root root       599 Jan  2 19:59 Dockerfile
drwxr-xr-x 8 uucp  143      4096 Sep  6 10:32 jdk1.8.0_151/
-rw-r--r-- 1 root root 189736377 Jan  2 19:54 jdk-8u151-linux-x64.tar.gz
drwxr-xr-x 3 root root      4096 Jan  2 19:55 mongodb-linux-x86_64-rhel70-3.2.9/
-rw-r--r-- 1 root root  72035914 Jan  2 19:53 mongodb-linux-x86_64-rhel70-3.2.9.tgz
-rw-r--r-- 1 root root       597 Jan  2 19:58 start_tomcat_and_mongo.sh
```

Step 3 Build an image.

docker build -t apptest .

Step 4 Upload the image to SWR. For details, see **Uploading an Image Through a Container Engine Client**.

----End

2.1.3.8 Creating a Container Workload

This section describes how to deploy a workload on CCE. When using CCE for the first time, create an initial cluster and add a node into the cluster.

NOTE

Containerized workloads are deployed in a similar way. The difference lies in:

- Whether environment variables need to be set.
- Whether cloud storage is used.

Required Cloud Services

- Cloud Container Engine (CCE): a highly reliable and high-performance service that allows enterprises to manage containerized applications. With support for Kubernetes-native applications and tools, CCE makes it simple to set up an environment for running containers in the cloud.
- Elastic Cloud Server (ECS): a scalable and on-demand cloud server. It helps you to efficiently set up reliable, secure, and flexible application environments, ensuring stable service running and improving O&M efficiency.
- Virtual Private Cloud (VPC): an isolated and private virtual network environment that users apply for in the cloud. You can configure the IP

address ranges, subnets, and security groups, as well as assign elastic IP addresses and allocate bandwidth in a VPC.

Basic Concepts

- A cluster is a collection of computing resources, including a group of node resources. A container runs on a node. Before creating a containerized application, you must have an available cluster.
- A node is a virtual or physical machine that provides computing resources. You must have sufficient node resources to ensure successful operations such as creating applications.
- A workload indicates a group of container pods running on CCE. CCE supports third-party application hosting and provides the full lifecycle (from deployment to O&M) management for applications. This section describes how to use a container image to create a workload.

Procedure

Step 1 Prepare the environment as described in **Table 2-5**.

Table 2-5 Preparing the environment

No.	Category	Procedure
1	Creating a VPC	Create a VPC before you create a cluster. A VPC provides an isolated, configurable, and manageable virtual network environment for CCE clusters. If you have a VPC already, skip to the next task. 1. Log in to the management console. 2. In the service list, choose **Networking** > **Virtual Private Cloud**. 3. On the **Dashboard** page, click **Create VPC**. 4. Follow the instructions to create a VPC. Retain default settings for parameters unless otherwise specified.

No.	Category	Procedure
2	Creating a key pair	Create a key pair before you create a containerized application. Key pairs are used for identity authentication during remote login to a node. If you have a key pair already, skip this task. 1. Log in to the management console. 2. In the service list, choose **Data Encryption Workshop** under **Security & Compliance**. 3. In the navigation pane, choose **Key Pair Service**. On the **Private Key Pairs** tab, click **Create Key Pair**. 4. Enter a key pair name, select **I agree to have the private key managed on the cloud** and **I have read and agree to the Key Pair Service Disclaimer**, and click **OK**. 5. In the dialog box displayed, click **OK**. View and save the key pair. For security purposes, a key pair can be downloaded only once. Keep it secure to ensure successful login.

Step 2 Create a cluster and a node.

1. Log in to the CCE console. On the **Clusters** page, click **Buy Cluster** and select the type for the cluster to be created.

   Configure cluster parameters and select the VPC created in **Step 1**. For details, see **Buying a CCE Cluster**.

2. Buy a node and select the key pair created in **Step 1** as the login option. For details, see **Creating a Node**.

Step 3 Deploy a workload on CCE.

1. Log in to the CCE console and click the name of the cluster to access the cluster console. In the navigation pane, choose **Workloads** and click **Create Workload**.
2. Configure the following parameters, and retain the default settings for other parameters:
   - **Workload Name**: Set it to **apptest**.
   - **Pods**: Set it to **1**.
3. In the **Container Settings** area, select the image uploaded in **Building and Uploading an Image**.
4. In the **Container Settings** area, choose **Environment Variables** and add environment variables for interconnecting with the MySQL database. The environment variables are set in the **startup script**.

   NOTE

   In this example, interconnection with the MySQL database is implemented through configuring the environment variables. Determine whether to use environment variables based on your service requirements.

Table 2-6 Configuring environment variables

Variable Name	Variable Value/Variable Reference
MYSQL_DB	Database name.
MYSQL_URL	IP address and port number of the database.
MYSQL_USER	Database username.
MYSQL_PASSWORD	Database user password.

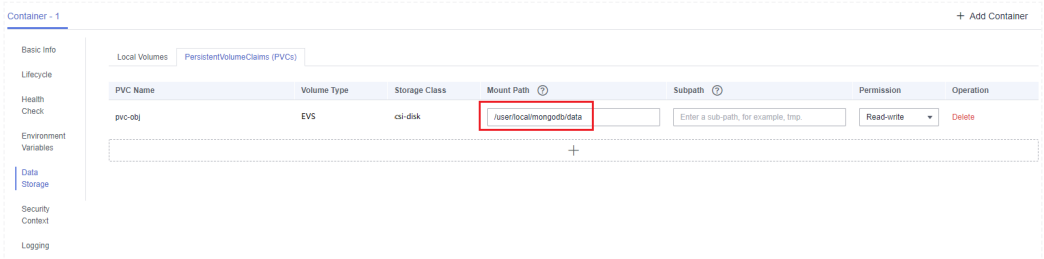
5. In the **Container Settings** area, choose **Data Storage** and configure cloud storage for persistent data storage.

   NOTE

   In this example, the MongoDB database is used and persistent data storage is also needed, so you need to configure cloud storage. Determine whether to use cloud storage based on your service requirements.

   The mounted path must be the same as the MongoDB storage path in the Docker startup script. For details, see the **startup script**. In this example, the path is **/usr/local/mongodb/data**.

   Figure 2-4 Configuring cloud storage

6. In the **Service Settings** area, click + to add a service, configure workload access parameters, and click **OK**.

   NOTE

   In this example, the application will be accessible from public networks by using an elastic IP address.

   - **Service Name**: name of the application that can be accessed externally. In this example, this parameter is set to **apptest**.
   - **Service Type**: In this example, select **NodePort**.
   - **Service Affinity**
     - **Cluster-level**: The IP addresses and access ports of all nodes in a cluster can be used to access the workload associated with the Service. Service access will cause performance loss due to route redirection, and the source IP address of the client cannot be obtained.
     - **Node-level**: Only the IP address and access port of the node where the workload is located can be used to access the workload

associated with the Service. Service access will not cause performance loss due to route redirection, and the source IP address of the client can be obtained.

- **Port**
  - **Protocol**: Set it to **TCP**.
  - **Service Port**: port for accessing the Service.
  - **Container Port**: port that the application will listen on the container. In this example, this parameter is set to **8080**.
  - **Node Port**: Set it to **Auto**. The system automatically opens a real port on all nodes in the current cluster and then maps the port number to the container port.

7. Click **Create Workload**.

   After the workload is created, you can view the running workload in the workload list.

----End

Verifying a Workload

After a workload is created, you can access the workload to check whether the deployment is successful.

In the preceding configuration, the NodePort mode is selected to access the workload by using **IP address:Port number**. If the access is successful, the workload is successfully deployed.

You can obtain the access mode from the **Access Mode** tab on the workload details page.

3 Migration

3.1 Migrating Container Images

3.1.1 Solution 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 three ways for migrating image repositories to SWR smoothly. You can select one as required.

Migration Solutions

Table 3-1 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.
Migrating Images to SWR Using image-migrator	A large number of images	• Many-to-many image repository synchronization is supported. • 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. • Flushing the Blob information of synchronized images avoids repetition. • The number of concurrent synchronization tasks can be adjusted in the configuration file. • Automatically retrying failed synchronization tasks can resolve most network jitter during image synchronization. • Docker or other programs are not required.

Solution	Application Scenario	Precautions
Synchronizing Images Across Clouds from Harbor to SWR	A customer deploys services in multiple clouds and uses Harbor as their image repository.	Only Harbor v1.10.5 and later versions are supported.

3.1.2 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 target container is located and log in to SWR. For details, see **Uploading an Image Through a Container Engine Client**.
2. Tag the images.

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

   Example:

   docker tag nginx:v1 swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/nginx:v1
3. Run the following command to push the images to the target image repository.

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

   Example:

   docker push swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/nginx:v1
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

3.1.3 Migrating Images to SWR Using image-migrator

To ensure that container images can be properly pulled after cluster migration and improve container deployment efficiency, you are advised to migrate self-built image repositories to Huawei Cloud SoftWare Repository for Container (SWR).

image-migrator is an image migration tool that can automatically migrate images from a Docker registry built on Docker Registry v2 to SWR.

Preparations

Before the migration, prepare a server with kubectl installed for the connection between the source and destination clusters. The server must have at least 5 GB local disk space and at least 8 GB memory so that image-migrator can work properly and can store related data, such as data collected from the source cluster and recommendation data of the destination cluster.

image-migrator can run on Linux (x86 and Arm) and Windows.

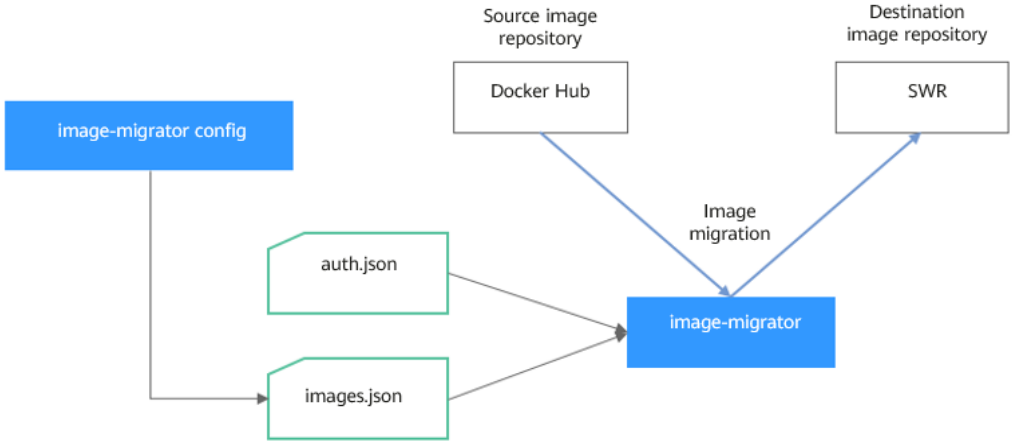
Before using image-migrator in Linux, run the **chmod u+x** *Tool name* command (for example, **chmod u+x image-migrator-linux-amd64**) to grant the execute permission.

Table 3-2 Obtaining the image-migrator package

image-migrator	image-migrator is an image migration tool that can automatically migrate images from a Docker registry built on Docker Registry v2 to SWR, or from a registry on a third-party cloud to SWR.	Linux x86: **https://ucs-migration.obs.cn-north-4.myhuaweicloud.com/toolkits/image-migrator-linux-amd64** Linux Arm: **https://ucs-migration.obs.cn-north-4.myhuaweicloud.com/toolkits/image-migrator-linux-arm64** Windows: **https://ucs-migration.obs.cn-north-4.myhuaweicloud.com/toolkits/image-migrator-windows-amd64.exe**

How image-migrator Works

Figure 3-1 How image-migrator works



When using image-migrator to migrate images to SWR, you need to prepare two files. One is the registry access permission file **auth.json**. The two objects in the file are the accounts and passwords of the source and destination registries. The other is the image list file **images.json**, which consists of multiple image synchronization rules. Each rule contains a source repository (key) and a destination repository (value). Place these two files in the directory where image-migrator is located and run a simple command to migrate the image. The two files are described as follows:

- **auth.json**

  auth.json is the registry access permission file. Each object is the username and password of a registry. Generally, you must have permission to pull images from and access image tags in the source registry and permission to push images to and create repositories in the destination registry. If you access a registry anonymously, you do not need to enter the username and password. Structure of the **auth.json** file:

  ```
  {
     "Source registry address": { },
     "Destination registry address": {
        "username": "xxxxxx",
        "password": "xxxxxx",
        "insecure": true
     }
  }
  ```

  To be more specific:

  – The *Source registry address* and *Destination registry address* can be in the *registry* or *registry/namespace* format matching that in **images.json**. The matched URL in images uses the corresponding username and password for image synchronization. The *registry/namespace* format is preferred.

    If the destination registry address is in the *registry* format, you can obtain it from the SWR console. On the **Dashboard** page, click **Generate Login Command** in the upper right corner. The domain name at the end of the login command is the SWR registry address, for example, **swr.cn-north-4.myhuaweicloud.com**. Note that the address varies depending on the region. Switch to the corresponding region to obtain the address. If

the value is in the *registry/namespace* format, replace *namespace* with the organization name of SWR.

- **username**: (Optional) username. You can set it to a specific value or use a string of the ${env} or $env type to reference an environment variable.
- **password**: (Optional) password. You can set it to a specific value or use a string of the ${env} or $env type to reference an environment variable.
- **insecure**: (Optional) whether *registry* is an HTTP service. If yes, the value of **insecure** is **true**. The default value is **false**.

NOTE

The username of the destination SWR registry is in the following format: *Regional project name*@*AK*. The password is the encrypted login key of the AK and SK. For details, see **Obtaining a Long-Term Valid Login Command**.

Example:

```
{
   "quay.io/coreos": { },
   "swr.cn-north-4.myhuaweicloud.com": {
      "username": "cn-north-4@RVHVMX******",
      "password": "cab4ceab4a1545***************",
      "insecure": true
   }
}
```

- **images.json**

  This file is essentially a list of images to migrate and consists of multiple image synchronization rules. Each rule contains a source repository (key) and a destination repository (value). The specific requirements are as follows:

  a. The largest unit that can be synchronized using one rule is repository. The entire namespace or registry cannot be synchronized using one rule.
  b. The formats of the source and destination repositories are similar to those of the image URL used by the **docker pull/push** command (*registry/namespace/repository:tag*).
  c. Both the source and destination repositories (if the destination registry is not an empty string) contain at least *registry*/*namespace*/*repository*.
  d. The source registry field cannot be empty. To synchronize data from a source registry to multiple destination registries, you need to configure multiple rules.
  e. The destination repository name can be different from the source repository name. In this case, the synchronization function is similar to docker pull + docker tag + docker push.
  f. If the source repository field does not contain tags, all tags of the repository have been synchronized to the destination repository. In this case, the destination repository cannot contain tags.
  g. If the source repository field contains tags, only one tag in the source repository has been synchronized to the destination repository. If the destination repository does not contain tags, the source tag is used by default.
  h. If the destination repository is an empty string, the source image will be synchronized to the default namespace of the default registry. The repository and tag are the same as those of the source repository. The default registry and namespace can be configured using command line parameters and environment variables.

Example:

```
{
    "quay.io/coreos/etcd:1.0.0": "swr.cn-north-4.myhuaweicloud.com/test/etcd:1.0.0",
    "quay.io/coreos/etcd": "swr.cn-north-4.myhuaweicloud.com/test/etcd",
    "quay.io/coreos/etcd:2.7.3": "swr.cn-north-4.myhuaweicloud.com/test/etcd"
}
```

We provide a config subcommand of the image-migrator tool to automatically obtain the image that is being used by the workload in the cluster. For details, see **Usage of image-migrator config**. After obtaining the **images.json** file, you can modify, add, or delete its content as needed.

How to Use image-migrator

NOTE

image-migrator can run on Linux (x86 and Arm) and Windows. The usage is similar in both environments. This section uses the Linux (x86) environment as an example.

If Linux (Arm) or Windows is used, replace **image-migrator-linux-amd64** in the following command with **image-migrator-linux-arm64** or **image-migrator-windows-amd64.exe**.

Run **./image-migrator-linux-amd64 -h** in the directory where image-migrator is located to learn about its usage.

- **--auth**: specifies the path of **auth.json**. By default, **auth.json** is stored in the directory where image-migrator is located.
- **--images**: specifies the path of **images.json**. By default, **images.json** is stored in the directory where image-migrator is located.
- **--log**: specifies the path for storing logs generated by image-migrator. The default value is **image-migrator.log** in the current directory of image-migrator.
- **--namespace**: specifies the default namespace of the destination repository. That is, if the namespace of the destination repository is not specified in **images.json**, you can specify it when running the migration command.
- **--registry**: specifies the default registry of the destination repository. That is, if the registry of the destination repository is not specified in **images.json**, you can specify it when running the migration command.
- **--retries**: specifies the number of retry times when the migration fails. The default value is **3**.
- **--workers**: specifies the number of concurrent workers for image migration. The default value is **7**.

```
$ ./image-migrator-linux-amd64 -h
A Fast and Flexible docker registry image images tool implement by Go.

Usage:
  image-migrator [flags]

Aliases:
  image-migrator, image-migrator

Flags:
      --auth string        auth file path. This flag need to be pair used with --images. (default "./auth.json")
  -h, --help               help for image-migrator
      --images string      images file path. This flag need to be pair used with --auth (default "./images.json")
      --log string         log file path (default "./image-migrator.log")
      --namespace string   default target namespace when target namespace is not given in the images
config file, can also be set with DEFAULT_NAMESPACE environment value
      --registry string    default target registry url when target registry is not given in the images config file,
```



```
can also be set with DEFAULT_REGISTRY environment value
  -r, --retries int        times to retry failed tasks (default 3)
  -w, --workers int        numbers of working goroutines (default 7)

$./image-migrator --workers=5 --auth=./auth.json --images=./images.json --namespace=test \
--registry=swr.cn-north-4.myhuaweicloud.com --retries=2
$ ./image-migrator
Start to generate images tasks, please wait ...
Start to handle images tasks, please wait ...
Images(38) migration finished, 0 images tasks failed, 0 tasks generate failed
```

Example:

./image-migrator --workers=5 --auth=./auth.json --images=./images.json --namespace=test --registry=swr.cn-north-4.myhuaweicloud.com --retries=2

The preceding command is used to migrate images in the **images.json** file to **swr.cn-north-4.myhuaweicloud.com/test**. If the migration fails, you can retry twice. A maximum of five images can be migrated at a time.

Usage of image-migrator config

The config subcommand of image-migrator can be used to obtain images used in cluster applications and generate the **images.json** file in the directory where the tool is located. You can run **./image-migrator-linux-amd64 config -h** to learn how to use the config subcommand.

- **-k, --kubeconfig**: specifies the location of the kubeconfig file of kubectl. The default value is **$HOME/.kube/config**. The kubeconfig file is used to configure access to the Kubernetes cluster. The kubeconfig file contains the authentication credentials and endpoints (access addresses) required for accessing and registering the Kubernetes cluster. For details, see the **Kubernetes documentation**.
- **-n, --namespaces**: specifies the namespace of the image to be obtained. Multiple namespaces are separated by commas (,), for example, ns1,ns2,ns3. The default value is **""**, indicating that images of all namespaces are obtained.
- **-t, --repo**: specifies the destination repository address (*registry*/*namespace*).

```
$ ./image-migrator-linux-amd64 config -h
generate images.json

Usage:
   image-migrator config [flags]

Flags:
   -h, --help               help for config
   -k, --kubeconfig string   The kubeconfig of k8s cluster's. Default is the $HOME/.kube/config. (default "/
root/.kube/config")
   -n, --namespaces string   Specify a namespace for information collection. If multiple namespaces are
specified, separate them with commas (,), such as ns1,ns2. default("") is all namespaces
   -t, --repo string         target repo,such as swr.cn-north-4.myhuaweicloud.com/test
```

Examples:

- Specify a namespace:

  ./image-migrator-linux-amd64 config -n default -t swr.cn-north-4.myhuaweicloud.com/test

- Specify multiple namespaces:

  ./image-migrator-linux-amd64 config -n default,kube-system -t swr.cn-north-4.myhuaweicloud.com/test

- If no namespace is specified, images of all namespaces are obtained:

  ./image-migrator-linux-amd64 config -t swr.cn-north-4.myhuaweicloud.com/test

Procedure

Step 1 Prepare the registry access permission file **auth.json**.

Create an **auth.json** file and modify it based on the format. If the repository is accessed anonymously, you do not need to enter information such as the username and password. Place the file in the directory where image-migrator is located.

Example:

```
{
    "quay.io/coreos": { },
    "swr.cn-north-4.myhuaweicloud.com": {
        "username": "cn-north-4@RVHVMX******",
        "password": "cab4ceab4a1545***************",
        "insecure": true
    }
}
```

For details about the parameters, see the **auth.json file**.

Step 2 Prepare the image list file **images.json**.

1. Connect to the source cluster using kubectl. For details, see **Connecting to a Cluster Using kubectl**.
2. Run the config subcommand for image migration to generate the **images.json** file.

   You can refer to the methods and examples in **Usage of image-migrator config** to obtain the image used in the source cluster application without specifying the namespace, or by specifying one or more namespaces.
3. Modify the **images.json** to make it meet the requirements in **images.json file**.

Step 3 Migrate images.

You can run the default **./image-migrator-linux-amd64** command to migrate images or configure image-migrator parameters as needed.

For example, run the following command:

./image-migrator-linux-amd64 --workers=5 --auth=./auth.json --images=./images.json --namespace=test --registry=swr.cn-north-4.myhuaweicloud.com --retries=2

Example:

```
$ ./image-migrator-linux-amd64
Start to generate images tasks, please wait ...
Start to handle images tasks, please wait ...
Images(38) migration finished, 0 images tasks failed, 0 tasks generate failed
```

Step 4 View the result.

After the preceding command is executed, information similar to the following is displayed:


```
Images(38) migration finished, 0 images tasks failed, 0 tasks generate failed
```

The preceding information indicates that 38 images have been migrated to SWR.

----End

3.1.4 Synchronizing Images Across Clouds from Harbor to SWR

Scenarios

A customer deploys services in multiple clouds and uses Harbor as their image repository. There are two scenarios for synchronizing images from Harbor to SWR:

1. Harbor accesses SWR through a public network. For details, see **Accessing SWR Through a Public Network**.
2. Harbor accesses SWR through a VPC endpoint by using a private line. For details, see **Accessing SWR Through a VPC Endpoint by Using a Private Line**.

Background

Harbor is an open source enterprise-class Docker Registry server developed by VMware. It extends the Docker Distribution by adding the functionalities such as role-based access control (RBAC), image scanning, and image replication. Harbor has been widely used to store and distribute container images.

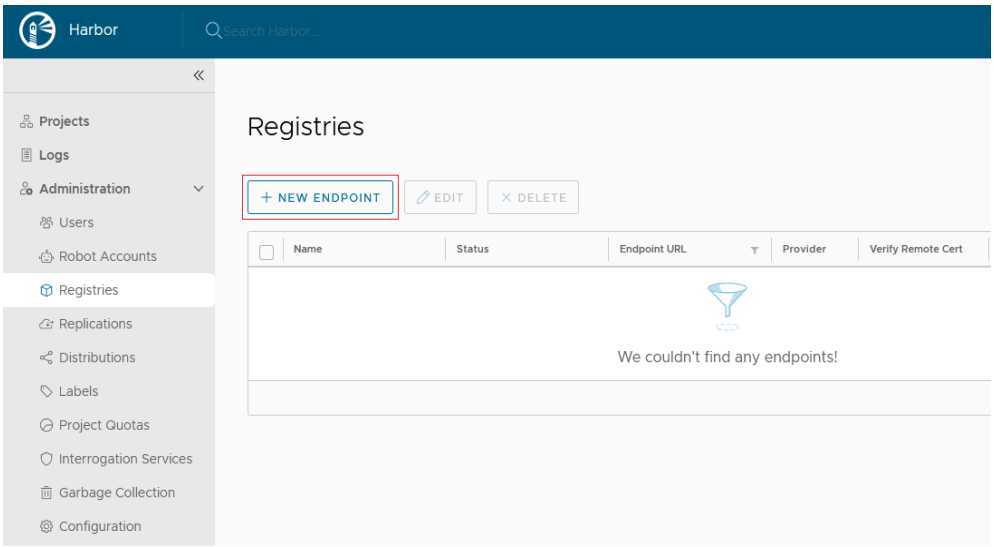
Accessing SWR Through a Public Network

Step 1 Configure a registry endpoint on Harbor.

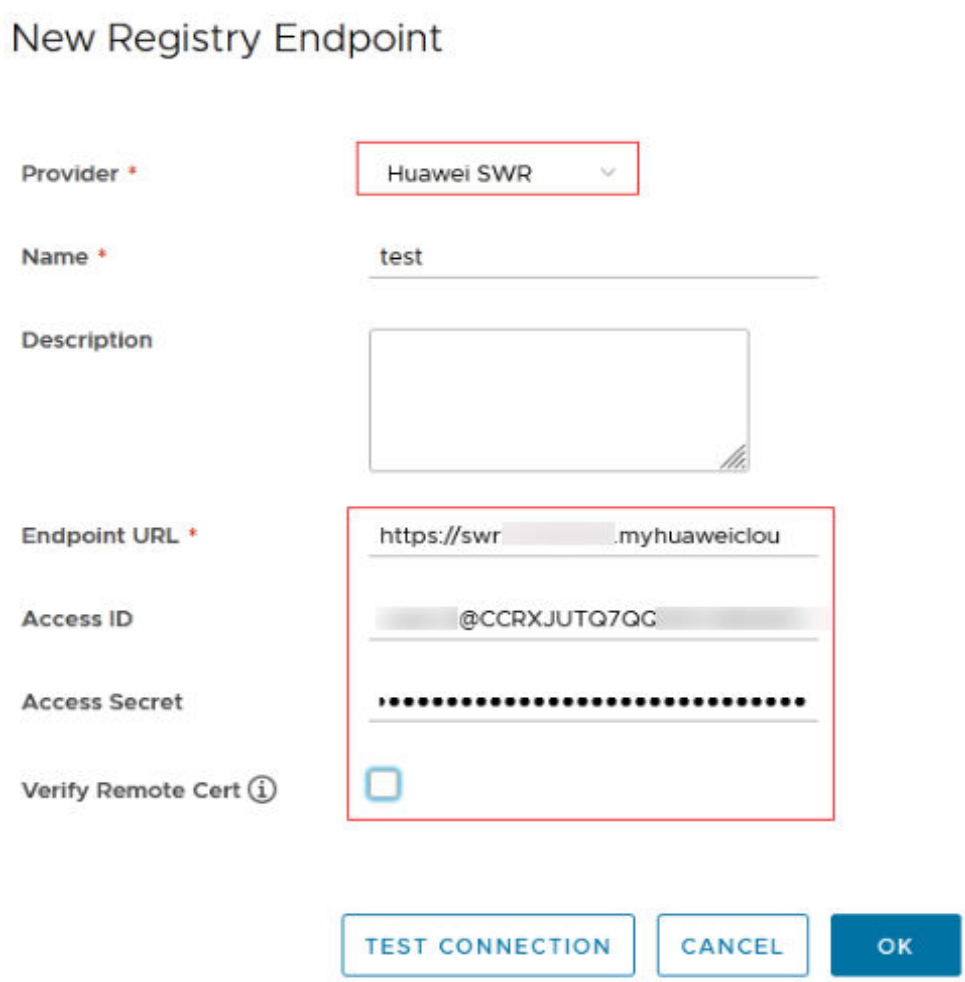
NOTE

Huawei Cloud SWR has integrated with Harbor 1.10.5 and later versions. You only need to set **Provider** to **Huawei SWR** when configuring your endpoint. This document uses Harbor 2.4.1 as an example.

1. Add an endpoint.



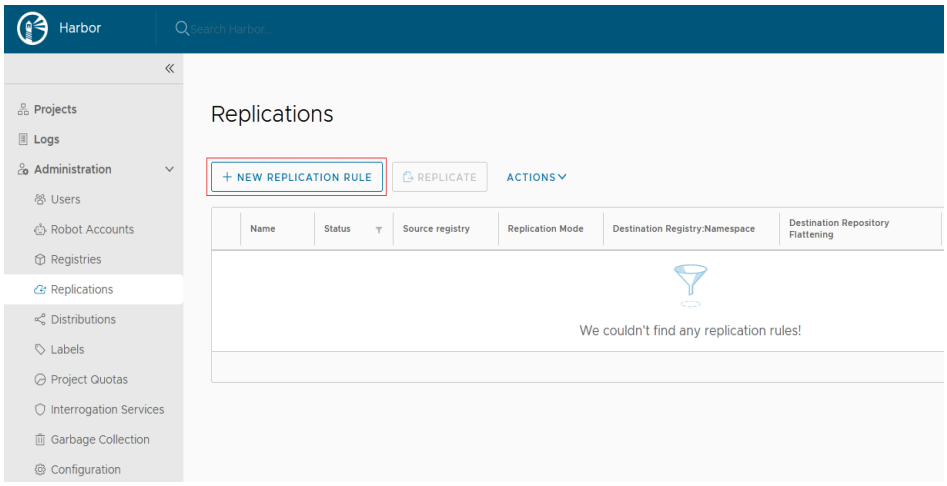
2. Configure the following parameters.



- **Provider**: Select **Huawei SWR**.
- **Name**: Enter a customized name.
- **Endpoint URL**: Enter the public network domain name of SWR in the format of **https://{*SWR image repository address*}**. To obtain the image repository address, log in to the SWR console, choose **My Images**, and click **Upload Through Client**. You can view the image repository address of the current region on the page that is displayed.
- **Access ID**: Enter an access ID in the format of **Regional project name@[AK]**.
- **Access Secret**: Enter an AK/SK. To obtain an AK/SK, see **Obtaining a Long-Term Valid Login Command**.
- **Verify Remote Cert**: **Deselect** the option (recommended).

Step 2 Configure a replication rule.

1. Create a replication rule.

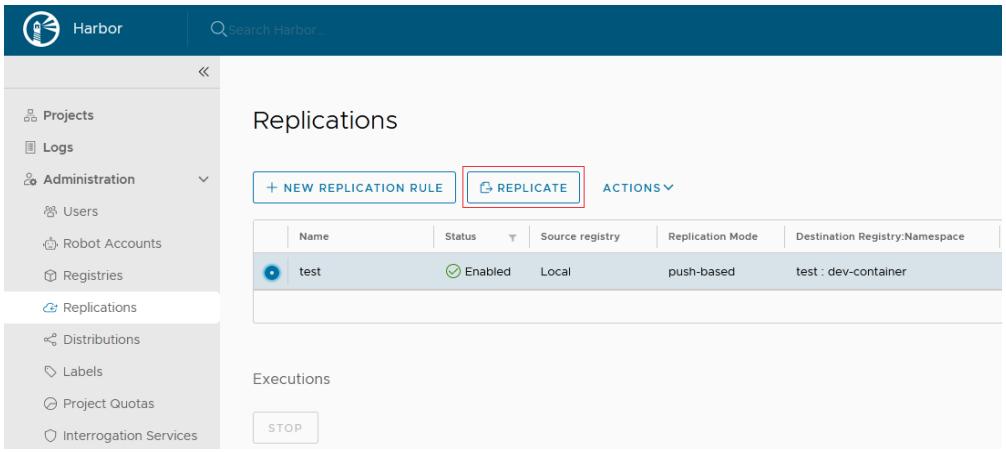


2. Configure the following parameters.
   - **Name**: Enter a customized name.
   - **Replication mode**: Select **Push-based**, indicating that images are pushed from the local Harbor to the remote repository.
   - **Source resource filter**: Filters images on Harbor based on the configured rules.
   - **Destination registry**: Select the endpoint created in **Step 1**.
   - **Destination**

     Namespace: Enter the organization name on SWR.

     Flattening: Select **Flatten All Levels**, indicating that the hierarchy of the registry is reduced when replicating images. If the directory of Harbor registry is **library/nginx** and the directory of the endpoint namespace is **dev-container**, after you flatten all levels, the directory of the endpoint namespace is **library/nginx -> dev-container/nginx**.
   - **Trigger Mode**: Select **Manual**.
   - **Bandwidth**: Set the maximum network bandwidth when executing the replication rule. The value **–1** indicates no limitation.

Step 3 After creating the replication rule, select it and click **REPLICATE** to complete the replication.



----End

Accessing SWR Through a VPC Endpoint by Using a Private Line

Step 1 Configure a VPC endpoint.

Step 2 Obtain the private network IP address and domain name of the VPC. (By default, the domain name resolution rule is automatically added to Huawei Cloud VPCs, so you only need to configure hosts for non-Huawei Cloud endpoints.) You can query the IP address and domain name in **Private Domain Name** on the VPC endpoint details page.

Step 3 Configure a registry endpoint on Harbor.

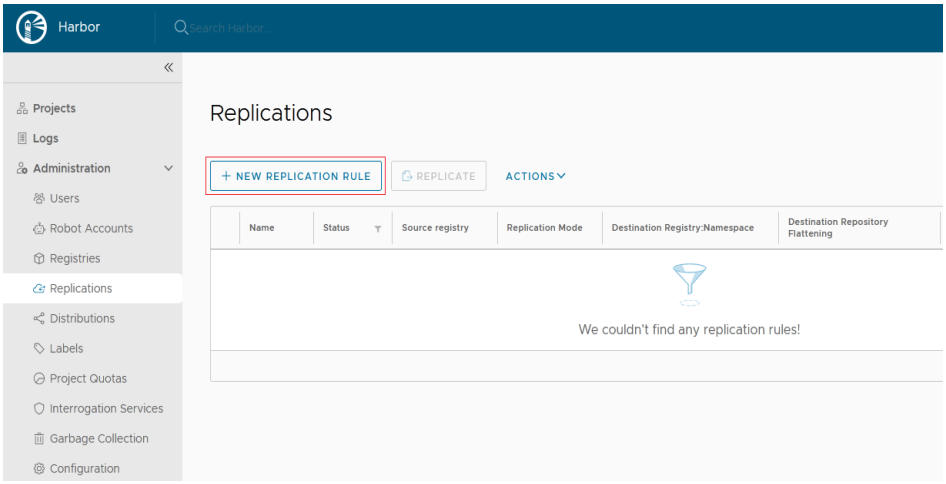
NOTE

Huawei Cloud SWR has integrated with Harbor 1.10.5 and later versions. You only need to set **Provider** to **Huawei SWR** when configuring your endpoint. This document uses Harbor 2.4.1 as an example.

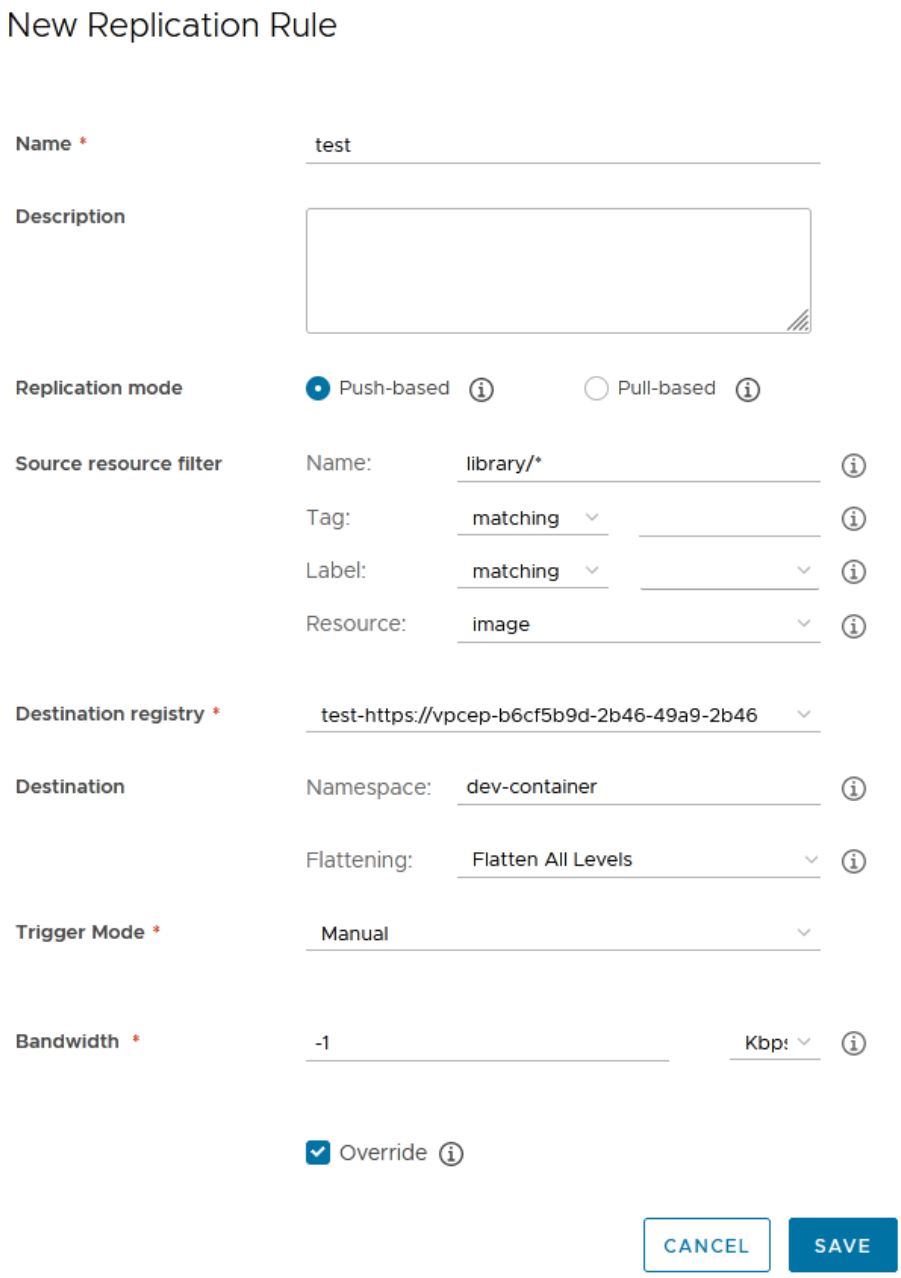
1. Add an endpoint.
2. Configure the following parameters.
   - **Provider**: Select **Huawei SWR**.
   - **Name**: Enter a customized name.
   - **Endpoint URL**: Enter **the private network domain name of the VPC endpoint**, which must start with **https**. In addition, the domain name mapping must be configured in the container where Harbor is located.
   - **Access ID**: Enter an access ID in the format of **Regional project name@[AK]**.
   - **Access Secret**: Enter an AK/SK. To obtain an AK/SK, see **Obtaining a Long-Term Valid Login Command**.
   - **Verify Remote Cert**: **Deselect** the option.

Step 4 Configure a replication rule.

1. Create a replication rule.



2. Configure the following parameters.



- **Name**: Enter a customized name.
- **Replication mode**: Select **Push-based**, indicating that images are pushed from the local Harbor to the remote repository.
- **Source resource filter**: Filters images on Harbor based on the configured rules.
- **Destination registry**: Select the endpoint created in **Step 3**.
- **Destination**

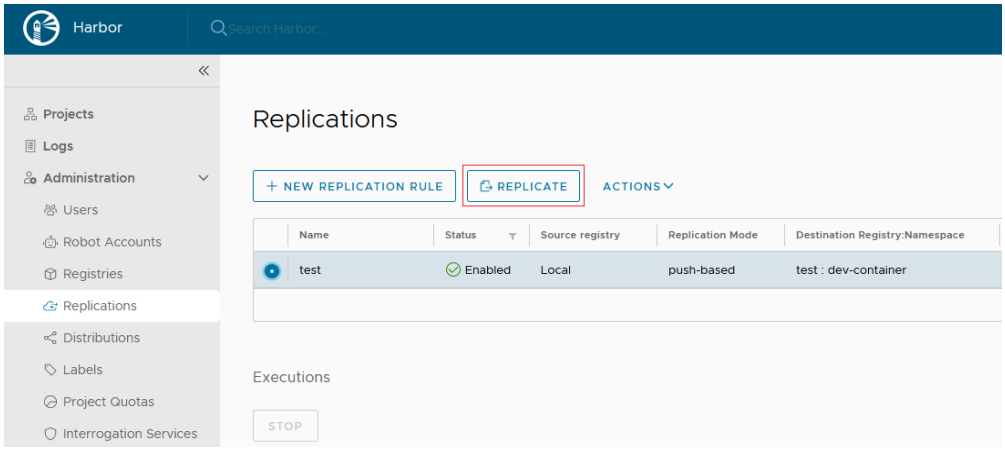
  Namespace: Enter the organization name on SWR.

  Flattening: Select **Flatten All Levels**, indicating that the hierarchy of the registry is reduced when replicating images. If the directory of Harbor registry is **library/nginx** and the directory of the endpoint namespace is

dev-container, after you flatten all levels, the directory of the endpoint namespace is **library/nginx -> dev-container/nginx**.

- **Trigger Mode**: Select **Manual**.
- **Bandwidth**: Set the maximum network bandwidth when executing the replication rule. The value **–1** indicates no limitation.

Step 5 After creating the replication rule, select it and click **REPLICATE** to complete the replication.



----End

3.2 Migrating Kubernetes Clusters to CCE

3.2.1 Solution Overview

Application Scenarios

Containers are growing in popularity and Kubernetes simplifies containerized deployment. Many companies 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. This increases the labor costs while decreasing the efficiency.

In terms of performance, an on-premises cluster has poor scalability due to its fixed specifications. Auto scaling cannot be implemented in case of traffic surges, which may easily result in the insufficient or waste of cluster resources. In addition, disaster recovery risks are not considered for deploying an on-premises cluster, leading to poor reliability. Once a fault occurs, the entire cluster may fail, resulting in serious production incidents.

Now you can address the preceding challenges by using CCE, a service that allows easy cluster management and flexible scaling, integrated with application service mesh and Helm charts to simplify cluster O&M and reduce operations costs. CCE is easy to use and delivers high performance, security, reliability, openness, and compatibility. This section describes the solution and procedure for migrating on-premises clusters to CCE.

Precautions

Compared with on-premises Kubernetes clusters, CCE clusters have multiple advantages. For details, see **Product Advantages**. There are some restrictions when using CCE clusters. For details, see **Notes and Constraints**. Evaluate the restrictions before using CCE clusters.

Migration Solution

This section describes a cluster migration solution, which applies to the following types of clusters:

- Kubernetes clusters built in local IDCs
- On-premises clusters built using multiple ECSs
- Cluster services provided by other cloud service providers
- CCE clusters that are no longer maintained and cannot be upgraded in place

Before the migration, analyze all resources in the source clusters and then determine the migration solution. Resources that can be migrated include resources inside and outside the clusters, as listed in the following table.

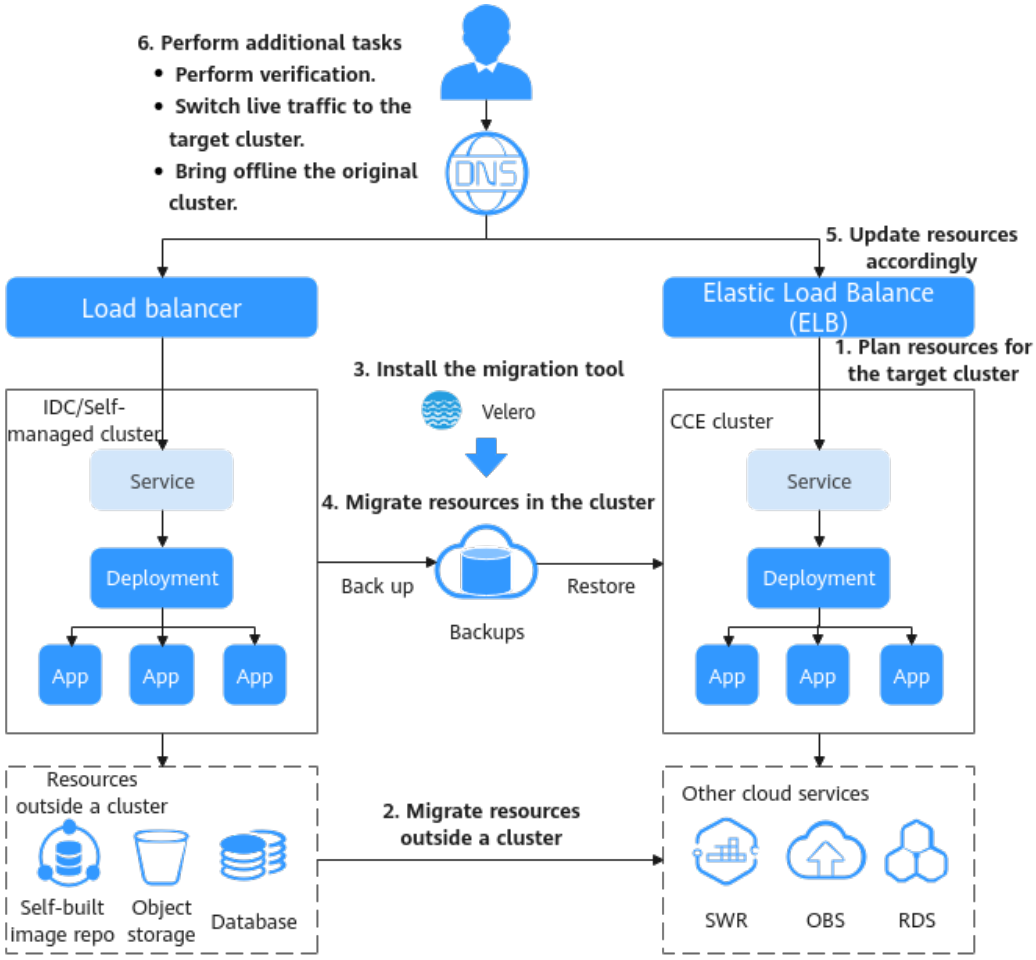
Table 3-3 Resources that can be migrated

Category	Migration Object	Remarks
Resources inside a cluster	All objects in a cluster, including pods, jobs, Services, Deployments, and ConfigMaps.	You are not advised to migrate the resources in the **velero** and **kube-system** namespaces. • **velero**: Resources in this namespace are created by the migration tool and do not need to be migrated. • **kube-system**: Resources in this namespace are system resources. If this namespace of the source cluster contains resources created by users, migrate the resources on demand. **CAUTION** If you are migrating or backing up cluster resources in CCE, for example, from a namespace to another, do not back up Secret **paas.elb**. It is because secret **paas.elb** is periodically updated. After the backup is complete, the secret may become invalid when it is restored. As a result, network storage functions are affected.

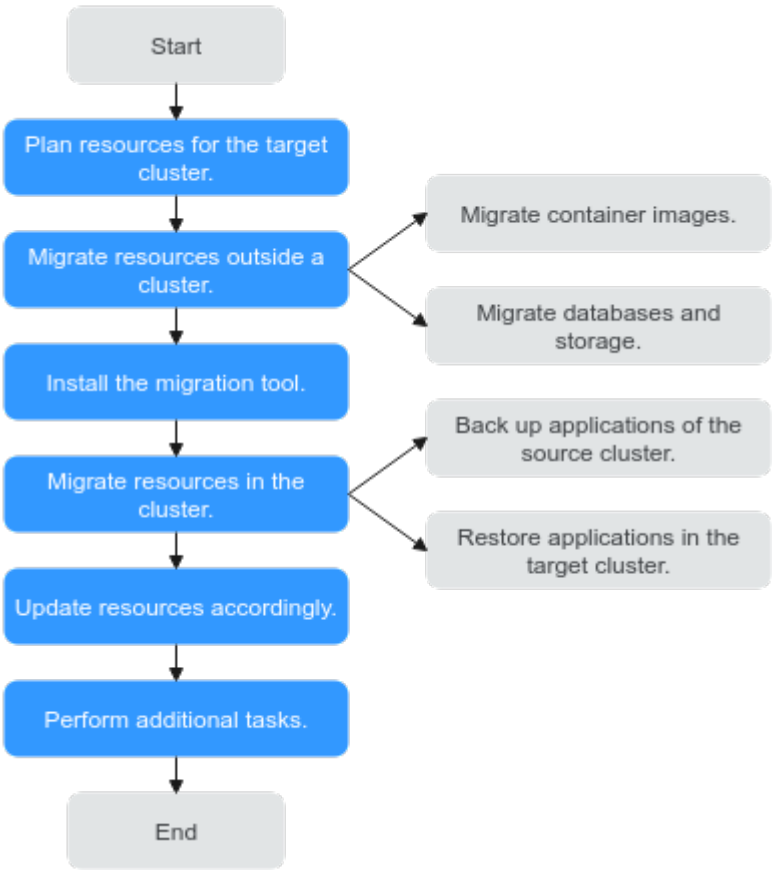
Category	Migration Object	Remarks
	PersistentVolumes (PVs) mounted to containers	Due to restrictions of the Restic tool, migration is not supported for the hostPath storage volume. For details about how to solve the problem, see **Storage Volumes of the HostPath Type Cannot Be Backed Up**.
Resources outside a cluster	On-premises image repository	Resources can be migrated to SoftWare Repository for Container (SWR).
	Non-containerized database	Resources can be migrated to Relational Database Service (RDS).
	Non-local storage, such as object storage	Resources can be migrated to Object Storage Service (OBS).

Figure 3-2 shows the migration process. You can migrate resources outside a cluster as required.

Figure 3-2 Migration solution diagram



Migration Process



The cluster migration process is as follows:

Step 1 Plan resources for the target cluster.

For details about the differences between CCE clusters and on-premises clusters, see **Key Performance Parameter** in **Planning Resources for the Target Cluster**. Plan resources as required and ensure that the performance configuration of the target cluster is the same as that of the source cluster.

Step 2 Migrate resources outside a cluster.

To migrate resources outside the cluster, see **Migrating Resources Outside a Cluster**.

Step 3 Install the migration tool.

After resources outside a cluster are migrated, you can use a migration tool to back up and restore application configurations in the source and target clusters. For details about how to install the tool, see **Installing the Migration Tool**.

Step 4 Migrate resources in the cluster.

Use Velero to back up resources in the source cluster to OBS and restore the resources in the target cluster. For details, see **Migrating Resources in a Cluster (Velero)**.

- **Backing Up Applications in the Source Cluster**

To back up resources, use the Velero tool to create a backup object in the original cluster, query and back up cluster data and resources, package the data, and upload the package to the object storage that is compatible with the S3 protocol. Cluster resources are stored in the JSON format.

- **Restoring Applications in the Target Cluster**

  During restoration in the target cluster, Velero specifies the temporary object bucket that stores the backup data, downloads the backup data to the new cluster, and redeploys resources based on the JSON file.

Step 5 Update resources accordingly.

After the migration, cluster resources may fail to be deployed. Update the faulty resources. The possible adaptation problems are as follows:

- **Updating Images**
- **Updating Services**
- **Updating the Storage Class**
- **Updating Databases**

Step 6 Perform additional tasks.

After cluster resources are properly deployed, verify application functions after the migration and switch service traffic to the target cluster. After confirming that all services are running properly, bring the source cluster offline.

----End

3.2.2 Planning Resources for the Target Cluster

CCE allows you to customize cluster resources to meet various service requirements. **Table 3-4** lists the key performance parameters of a cluster and provides the planned values. You can set the parameters based on your service requirements. It is recommended that the performance configuration be the same as that of the source cluster.

NOTICE

After a cluster is created, the resource parameters marked with asterisks (*) in **Table 3-4** cannot be modified.

Table 3-4 CCE cluster planning

Resource	Key Performance Parameter	Description	Example Value
Cluster	*Cluster Type	• **CCE cluster**: supports VM nodes. You can run your containers in a secure and stable container runtime environment based on a high-performance network model. • **CCE Turbo cluster**: runs on a cloud native infrastructure that features software-hardware synergy to support passthrough networking, high security and reliability, and intelligent scheduling, and BMS nodes.	CCE cluster
	*Network Model	• **VPC network**: The container network uses VPC routing to integrate with the underlying network. This network model is applicable to performance-intensive scenarios. The maximum number of nodes allowed in a cluster depends on the route quota in a VPC network. • **Tunnel network**: The container network is an overlay tunnel network on top of a VPC network and uses the VXLAN technology. This network model is applicable when there is no high requirements on performance. • **Cloud Native Network 2.0**: The container network deeply integrates the elastic network interface (ENI) capability of VPC, uses the VPC CIDR block to allocate container addresses, and supports passthrough networking to containers through a load balancer.	VPC network
	*Number of master nodes	• **3**: Three master nodes will be created to deliver better DR performance. If one master node is faulty, the cluster can still be available without affecting service functions. • **1**: A single master node will be created. This mode is not recommended in commercial scenarios.	3

Resource	Key Performance Parameter	Description	Example Value
Node	OS	• EulerOS • CentOS • Ubuntu	EulerOS

Resource	Key Performance Parameter	Description	Example Value
	Node Specifications (vary depending on the actual region)	• **General-purpose**: provides a balance of computing, memory, and network resources. It is a good choice for many applications. General-purpose nodes can be used for web servers, workload development, workload testing, and small-scale databases. • **Memory-optimized**: provides higher memory capacity than general-purpose nodes and is suitable for relational databases, NoSQL, and other workloads that are both memory-intensive and data-intensive. • **General computing-basic**: provides a balance of computing, memory, and network resources and uses the vCPU credit mechanism to ensure baseline computing performance. Nodes of this type are suitable for applications requiring burstable high performance, such as light-load web servers, enterprise R&D and testing environments, and low- and medium-performance databases. • **GPU-accelerated**: provides powerful floating-point computing and is suitable for real-time, highly concurrent massive computing. Graphical processing units (GPUs) of P series are suitable for deep learning, scientific computing, and CAE. GPUs of G series are suitable for 3D animation rendering and CAD. GPU-accelerated nodes can be added only to clusters of v1.11 or later. • **High-performance computing**: provides stable and ultra-high computing performance and is suitable for scientific computing and workloads that demand ultra-high computing power and throughput. • **General computing-plus**: provides stable performance and exclusive resources to enterprise-class workloads with high and stable computing performance.	General-purpose (node specifications: 4 vCPUs and 8 GiB memory)

Resource	Key Performance Parameter	Description	Example Value
		• **Disk-intensive**: supports local disk storage and provides high networking performance. It is designed for workloads requiring high throughput and data switching, such as big data workloads. • **Ultra-high I/O**: delivers ultra-low SSD access latency and ultra-high IOPS performance. This type of specifications is ideal for high-performance relational databases, NoSQL databases (such as Cassandra and MongoDB), and Elasticsearch. • **Ascend-accelerated**: Ascend-accelerated nodes powered by HiSilicon Ascend 310 AI processors are applicable to scenarios such as image recognition, video processing, inference computing, and machine learning.	
	System Disk	• **High I/O**: The backend storage media is SAS disks. • **Ultra-high I/O**: The backend storage media is SSD disks.	High I/O

Resource	Key Performance Parameter	Description	Example Value
	Storage Type	• **EVS volumes**: Mount an EVS volume to a container path. When containers are migrated, the attached EVS volumes are migrated accordingly. This storage mode is suitable for data that needs to be permanently stored. • **SFS volumes**: Create SFS volumes and mount them to a container path. The file system volumes created by the underlying SFS service can also be used. SFS volumes are applicable to persistent storage for frequent read/write in multiple workload scenarios, including media processing, content management, big data analysis, and workload analysis. • **OBS volumes**: Create OBS volumes and mount them to a container path. OBS volumes are applicable to scenarios such as cloud workload, data analysis, content analysis, and hotspot objects. • **SFS Turbo volumes**: Create SFS Turbo volumes and mount them to a container path. SFS Turbo volumes are fast, on-demand, and scalable, which makes them suitable for DevOps, containerized microservices, and enterprise office applications.	EVS volumes

3.2.3 Procedure

3.2.3.1 Migrating Resources Outside a Cluster

If your migration does not involve resources outside a cluster listed in **Table 3-3** or you do not need to use other services to update resources after the migration, skip this section.

Migrating Container Images

To ensure that container images can be properly pulled after cluster migration and improve container deployment efficiency, you are advised to migrate private images to SoftWare Repository for Container (SWR). CCE works with SWR to provide a pipeline for automated container delivery. Images are pulled in parallel, which greatly improves container delivery efficiency.

Manually migrate container images.

Step 1 Remotely log in to any node in the source cluster and run the **docker pull** command to pull all images to the local host.

Step 2 Log in to the SWR console, click **Login Command** in the upper right corner of the page, and copy the command.

Step 3 Run the copied login command on the node.

The message "Login Succeeded" will be displayed upon a successful login.

Step 4 Add tags to all local images.

```
docker tag [Image name 1:tag 1] [Image repository address]/[Organization name]/[Image name 2:tag 2]
```

- *[Image name 1:tag 1]*: name and tag of the local image to be pulled.
- *[Image repository address]*: You can obtain the image repository address on the SWR console.
- *[Organization name]*: Enter the name of the organization you created on the SWR console.
- *{Image name 2:Tag 2}*: image name and tag displayed on the SWR console.

The following is an example:

```
docker tag nginx:v1 swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/mynginx:v1
```

Step 5 Run the **docker push** command to upload all local container image files to SWR.

```
docker push [Image repository address]/[Organization name]/[Image name 2:tag 2]
```

The following is an example:

```
docker push swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/mynginx:v1
```

----End

Migrating Databases and Storage (On-Demand)

You can determine whether to use **Relational Database Service (RDS)** and **Object Storage Service (OBS)** based on your production requirements. After the migration is complete, reconfigure the database and storage for applications in the target CCE cluster.

Database migration

If your database is deployed without using containers and needs to be migrated to the cloud, **Data Replication Service (DRS)** can help you do this. DRS provides multiple capabilities, including online migration, backup migration, real-time disaster recovery, data synchronization, and data subscription. Contact O&M or development personnel to migrate the database. For details, see **Migrating Databases Across Cloud Platforms**. After the migration is complete, interconnect with the database by following the procedure described in **Updating Databases**.

Storage migration

If your cluster has connected to an object storage service and needs to be migrated to the cloud, **Object Storage Migration Service (OMS)** can help you migrate data to OBS. Other storage classes are not supported by official tools.

Contact O&M or development personnel to migrate object storage data. For details, see **Creating a Migration Task**. After the migration is complete, attach

the object storage to the application by referring to **Using an Existing OBS Bucket Through a Static PV**.

NOTE

Currently, you can use OMS to migrate object storage data from AWS China, Alibaba Cloud, Microsoft Azure, Baidu Cloud, Kingsoft Cloud, Ucloud, QingCloud, Qiniu Cloud, and Tencent Cloud to Huawei Cloud **OBS**.

3.2.3.2 Installing the Migration Tool

Velero is an open-source backup and migration tool for Kubernetes clusters. It integrates the persistent volume (PV) data backup capability of the Restic tool and can be used to back up Kubernetes resource objects (such as Deployments, jobs, Services, and ConfigMaps) in the source cluster. Data in the PV mounted to the pod is backed up and uploaded to the object storage. When a disaster occurs or migration is required, the target cluster can use Velero to obtain the corresponding backup data from OBS and restore cluster resources as required.

According to **Migration Solution**, prepare temporary object storage to store backup files before the migration. Velero supports OSB or **MinIO** as the object storage. OBS requires sufficient storage space for storing backup files. You can estimate the storage space based on your cluster scale and data volume. You are advised to use OBS for backup. For details about how to deploy Velero, see **Installing Velero**.

Prerequisites

- The Kubernetes version of the source on-premises cluster must be 1.10 or later, and the cluster can use DNS and Internet services properly.
- If you use OBS to store backup files, obtain the AK/SK of a user who has the right to operate OBS. For details, see **Obtaining Access Keys (AK/SK)**.
- If you use MinIO to store backup files, bind an EIP to the server where MinIO is installed and enable the API and console port of MinIO in the security group.
- The target CCE cluster has been created.
- The source cluster and target cluster must each have at least one idle node. It is recommended that the node specifications be 4 vCPUs and 8 GiB memory or higher.

(Optional) Installing MinIO

MinIO is an open-source, high-performance object storage tool compatible with the S3 API protocol. If MinIO is used to store backup files for cluster migration, you need a temporary server to deploy MinIO and provide services for external systems. If you use OBS to store backup files, skip this section and go to **Installing Velero**.

MinIO can be installed in any of the following locations:

- Temporary ECS outside the cluster

  If the MinIO server is installed outside the cluster, backup files will not be affected when a catastrophic fault occurs in the cluster.

- Idle nodes in the cluster

  You can remotely log in to a node to install MinIO or install the containerized MinIO. For details, see **Velero official document**.

  NOTICE

  For example, to install MinIO in a container, run the following command:

  - The storage type in the YAML file provided by Velero is **emptyDir**. You are advised to change the storage type to **HostPath** or **Local**. Otherwise, backup files will be permanently lost after the container is restarted.
  - Ensure that the MinIO service is accessible externally. Otherwise, backup files cannot be downloaded outside the cluster. You can change the Service type to NodePort or use other types of public network access Services.

Regardless of which deployment method is used, the server where MinIO is installed must have sufficient storage space, an EIP must be bound to the server, and the MinIO service port must be enabled in the security group. Otherwise, backup files cannot be uploaded or downloaded.

In this example, MinIO is installed on a temporary ECS outside the cluster.

Step 1 Download MinIO.

```
mkdir /opt/minio
mkdir /opt/miniodata
cd /opt/minio
wget https://dl.minio.io/server/minio/release/linux-amd64/minio
chmod +x minio
```

Step 2 Set the username and password of MinIO.

The username and password set using this method are temporary environment variables and must be reset after the service is restarted. Otherwise, the default root credential **minioadmin:minioadmin** will be used to create the service.

```
export MINIO_ROOT_USER=minio
export MINIO_ROOT_PASSWORD=minio123
```

Step 3 Create a service. In the command, **/opt/miniodata/** indicates the local disk path for MinIO to store data.

The default API port of MinIO is 9000, and the console port is randomly generated. You can use the **--console-address** parameter to specify a console port.

```
./minio server /opt/miniodata/ --console-address ":30840" &
```

NOTE

Enable the API and console ports in the firewall and security group on the server where MinIO is to be installed. Otherwise, access to the object bucket will fail.

Step 4 Use a browser to access http://{*EIP of the node where MinIO resides*}:30840. The MinIO console page is displayed.

----End

Installing Velero

Go to the OBS console or MinIO console and create a bucket named **velero** to store backup files. You can custom the bucket name, which must be used when

installing Velero. Otherwise, the bucket cannot be accessed and the backup fails. For details, see **Step 5**.

NOTICE

- Velero instances need to be installed and deployed in both the **source and target clusters**. The installation procedures are the same, which are used for backup and restoration, respectively.
- The master node of a CCE cluster does not provide a port for remote login. You can install Velero using kubectl.
- If there are a large number of resources to back up, you are advised to adjust the CPU and memory resources of Velero and node-agent to 1 vCPU and 1 GiB memory or higher. For details, see **Backup Tool Resources Are Insufficient**.
- The object storage bucket for storing backup files must be **empty**.

Download the latest, stable binary file from **https://github.com/vmware-tanzu/velero/releases**. This section uses Velero 1.13.1 as an example. The installation process in the source cluster is the same as that in the target cluster.

Step 1 Log in to a VM that can access the public network and use kubectl to access the cluster where Velero is to be installed.

Step 2 Download the binary file of Velero 1.13.1.

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.13.1/velero-v1.13.1-linux-amd64.tar.gz
```

Step 3 Install the Velero client.

```
tar -xvf velero-v1.13.1-linux-amd64.tar.gz
cp ./velero-v1.13.1-linux-amd64/velero /usr/local/bin
```

Step 4 Create the access key file **credentials-velero** for the backup object storage.

```
vim credentials-velero
```

Replace the AK/SK in the file based on the site requirements. When you use OBS, you can obtain the AK/SK by referring to **Obtaining Access Keys (AK/SK)**. If MinIO is used, the AK/SK are the username and password created in **Step 2**.

```
[default]
aws_access_key_id = {AK}
aws_secret_access_key = {SK}
```

Step 5 Deploy the Velero server. Change the value of **--bucket** to the name of the created object storage bucket. In this example, the bucket name is **velero**. For more information about custom installation parameters, see **Customize Velero Install**.

```
velero install \
  --provider aws \
  --plugins velero/velero-plugin-for-aws:v1.9.1 \
  --bucket velero \
  --secret-file ./credentials-velero \
  --use-node-agent \
  --use-volume-snapshots=false \
  --backup-location-config region=ap-southeast-1,s3ForcePathStyle="true",s3Url=http://obs.ap-southeast-1.myhuaweicloud.com
```


Table 3-5 Installation parameters of Velero

Parameter	Description
--provider	AWS S3 component to be used
--plugins	API component compatible with AWS S3. Both OBS and MinIO support the S3 protocol.
--bucket	Name of the object storage bucket for storing backup files. The bucket must be created in advance.
--secret-file	Secret file for accessing the object storage, that is, the **credentials-velero** file created in **Step 4**.
--use-node-agent	Whether to enable PV data backup. You are advised to enable this function. Otherwise, storage volume resources cannot be backed up.
--use-volume-snapshots	Whether to create the VolumeSnapshotLocation object for PV snapshot, which requires support from the snapshot program. Set this parameter to **false**.
--backup-location-config	OBS bucket configurations, including region, s3ForcePathStyle, and s3Url.
region	Region to which object storage bucket belongs. • If OBS is used, set this parameter according to your region, for example, **ap-southeast-1**. • If MinIO is used, set this parameter to **minio**.
s3ForcePathStyle	The value **true** indicates that the S3 file path format is used.
s3Url	API access address of the object storage bucket. • If OBS is used, set this parameter to **http://obs.*{region}*.myhuaweicloud.com** (*region* indicates the region where the object storage bucket is located). For example, if the region is Hong Kong (ap-southeast-1), the value is **http://obs.ap-southeast-1.myhuaweicloud.com**. • If MinIO is used, set this parameter to **http://*{EIP of the node where minio is located}*:9000**. The value of this parameter is determined based on the IP address and port of the node where MinIO is installed. **NOTE** – The access port in s3Url must be set to the API port of MinIO instead of the console port. The default API port of MinIO is 9000. – To access MinIO installed outside the cluster, enter the public IP address of MinIO.

Step 6 By default, a namespace named **velero** is created for the Velero instance. Run the following command to view the pod status:


```
$ kubectl get pod -n velero
NAME                    READY   STATUS    RESTARTS   AGE
node-agent-rn29c        1/1     Running   0          16s
velero-c9ddd56-tkzpk    1/1     Running   0          16s
```

NOTE

To prevent memory insufficiency during backup in the actual production environment, you are advised to change the CPU and memory allocated to node-agent and Velero by referring to **Backup Tool Resources Are Insufficient**.

Step 7 Check the interconnection between Velero and the object storage and ensure that the status is **Available**.

```
$ velero backup-location get
NAME      PROVIDER   BUCKET/PREFIX   PHASE       LAST VALIDATED                  ACCESS MODE   DEFAULT
default   aws        velero          Available   2021-10-22 15:21:12 +0800 CST   ReadWrite     true
```

----End

3.2.3.3 Migrating Resources in a Cluster (Velero)

Application Scenarios

WordPress is used as an example to describe how to migrate an application from an on-premises Kubernetes cluster to a CCE cluster. The WordPress application consists of the WordPress and MySQL components, which are containerized. The two components are bound to two local storage volumes of the Local type respectively and provide external access through the NodePort Service.

Before the migration, use a browser to access the WordPress site, create a site named **Migrate to CCE**, and publish an article to verify the integrity of PV data after the migration. The article published in WordPress will be stored in the **wp_posts** table of the MySQL database. If the migration is successful, all contents in the database will be migrated to the new cluster. You can verify the PV data migration based on the migration result.

Prerequisites

- Before the migration, clear the abnormal pod resources in the source cluster. If the pod is in the abnormal state and has a PVC mounted, the PVC is in the pending state after the cluster is migrated.
- Ensure that the cluster on the CCE side does not have the same resources as the cluster to be migrated because Velero does not restore the same resources by default.
- To ensure that container images can be properly pulled after cluster migration, migrate the images to SWR.
- CCE does not support EVS disks of the **ReadWriteMany** type. If resources of this type exist in the source cluster, change the storage type to **ReadWriteOnce**.
- Velero cannot back up or restore HostPath volumes. For details, see **Limitations**. To back up storage volumes of this type, replace the hostPath volumes with local volumes by referring to**Storage Volumes of the HostPath Type Cannot Be Backed Up**. If a backup task involves storage of the HostPath type, the storage volumes of this type will be automatically skipped and a warning message will be generated. This will not cause a backup failure.

Backing Up Applications in the Source Cluster

Step 1 (Optional) To back up the data of a specified storage volume in the pod, add an annotation to the pod. The annotation template is as follows:

```
kubectl -n <namespace> annotate <pod/pod_name> backup.velero.io/backup-volumes=<volume_name_1>,<volume_name_2>,...
```

- **<namespace>**: namespace where the pod is located.
- **<pod_name>**: pod name.
- **<volume_name>**: name of the persistent volume mounted to the pod. You can run the **describe** statement to query the pod information. The **Volume** field indicates the names of all persistent volumes attached to the pod.

Add annotations to the pods of WordPress and MySQL. The pod names are **wordpress-758fbf6fc7-s7fsr** and **mysql-5ffdfbc498-c45lh**. As the pods are in the default namespace **default**, the **-n <NAMESPACE>** parameter can be omitted.

```
kubectl annotate pod/wordpress-758fbf6fc7-s7fsr backup.velero.io/backup-volumes=wp-storage
kubectl annotate pod/mysql-5ffdfbc498-c45lh backup.velero.io/backup-volumes=mysql-storage
```

Step 2 Back up the application. During the backup, you can specify resources based on parameters. If no parameter is added, the entire cluster resources are backed up by default. For details about the parameters, see **Resource filtering**.

- **--default-volumes-to-fs-backup**: indicates that the PV backup tool is used to back up all storage volumes attached to a pod. HostPath volumes are not supported. If this parameter is not specified, the storage volume specified by annotation in **Step 1** is backed up by default. This parameter is available only when **--use-node-agent** is specified during **Velero installation**.
  ```
  velero backup create <backup-name> --default-volumes-to-fs-backup
  ```
- **--include-namespaces**: backs up resources in a specified namespace.
  ```
  velero backup create <backup-name> --include-namespaces <namespace>
  ```
- **--include-resources**: backs up the specified resources.
  ```
  velero backup create <backup-name> --include-resources deployments
  ```
- **--selector**: backs up resources that match the selector.
  ```
  velero backup create <backup-name> --selector <key>=<value>
  ```

In this section, resources in the namespace **default** are backed up. **wordpress-backup** is the backup name. Specify the same backup name when restoring applications. An example is as follows:

```
velero backup create wordpress-backup --include-namespaces default --default-volumes-to-fs-backup
```

If the following information is displayed, the backup task is successfully created:

```
Backup request "wordpress-backup" submitted successfully.
Run `velero backup describe wordpress-backup` or `velero backup logs wordpress-backup` for more details.
```

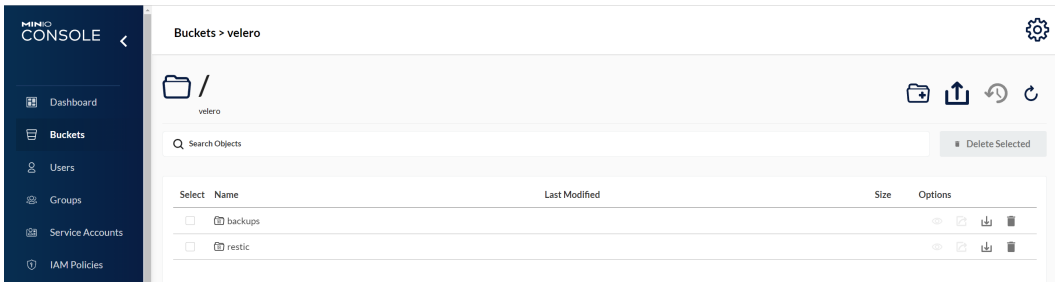
Step 3 Check the backup status.

```
velero backup get
```

Information similar to the following is displayed:

```
NAME              STATUS      ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
wordpress-backup  Completed   0        0          2021-10-14 15:32:07 +0800 CST   29d       default            <none>
```

In addition, you can go to the object bucket to view the backup files. The backups path is the application resource backup path, and the other is the PV data backup path.



----End

Restoring Applications in the Target Cluster

The storage infrastructure of an on-premises cluster is different from that of a cloud cluster. After the cluster is migrated, PVs cannot be mounted to pods. Therefore, during the migration, update the storage class of the target cluster to shield the differences of underlying storage interfaces between the two clusters when creating a workload and request storage resources of the corresponding type. For details, see **Updating the Storage Class**. If you migrate storage by using Object Storage Migration Service (OMS), you can mount object storage buckets to pods by referring to **Using an Existing OBS Bucket Through a Static PV**.

Step 1 Use kubectl to connect to the CCE cluster. Create a storage class with the same name as that of the source cluster.

In this example, the storage class name of the source cluster is **local** and the storage type is local disk. Local disks completely depend on the node availability. The data DR performance is poor. When the node is unavailable, the existing storage data is affected. Therefore, EVS volumes are used as storage resources in CCE clusters, and SAS disks are used as backend storage media.

NOTE

- When an application containing PV data is restored in a CCE cluster, the defined storage class dynamically creates and mounts storage resources (such as EVS volumes) based on the PVC.
- The storage resources of the cluster can be changed as required, not limited to EVS volumes. To mount other types of storage, such as file storage and object storage, see **Updating the Storage Class**.

YAML file of the migrated cluster:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

The following is an example of the YAML file of the migration cluster:

```
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-disk
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
```



```
  everest.io/disk-volume-type: SAS
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Step 2 Use the Velero tool to create a restore and specify a backup named **wordpress-backup** to restore the WordPress application to the CCE cluster.

```
velero restore create --from-backup wordpress-backup
```

You can run the **velero restore get** statement to view the application restoration status.

Step 3 After the restoration is complete, check whether the application is running properly. If other adaptation problems may occur, rectify the fault by following the procedure described in **Updating Resources Accordingly**.

----End

3.2.3.4 Updating Resources Accordingly

Updating Images

The WordPress and MySQL images used in this example can be pulled from SWR. Therefore, the image pull failure (ErrImagePull) will not occur. If the application to be migrated is created from a private image, perform the following steps to update the image:

Step 1 Migrate the image resources to SWR. For details, see **Uploading an Image Through a Container Engine Client**.

Step 2 Log in to the SWR console and obtain the image path used after the migration.

The image path is in the following format:

```
'swr.{Region}.myhuaweicloud.com/{Organization name}/{Image name}:{Tag name}'
```

Step 3 Run the following command to modify the workload and replace the **image** field in the YAML file with the image path:

```
kubectl edit deploy wordpress
```

Step 4 Check the running status of the workload.

----End

Updating Services

After the cluster is migrated, the Service of the source cluster may fail to take effect. You can perform the following steps to update the Service. If ingresses are configured in the source cluster, connect the new cluster to ELB again after the migration. For details, see **Using kubectl to Create an ELB Ingress**.

Step 1 Connect to the cluster using kubectl.

Step 2 Edit the YAML file of the corresponding Service to change the Service type and port number.

```
kubectl edit svc wordpress
```


To update load balancer resources, connect to ELB again. Add the annotations by following the procedure described in **Creating an Ingress - Interconnecting with an Existing Load Balancer**.

```
annotations:
  kubernetes.io/elb.class: union # Shared load balancer
  kubernetes.io/elb.id: 9d06a39d-xxxx-xxxx-xxxx-c204397498a3    # Load balancer ID, which can be queried on the ELB console.
  kubernetes.io/elb.subnet-id: f86ba71c-xxxx-xxxx-xxxx-39c8a7d4bb36    # ID of the cluster where the subnet resides
  kubernetes.io/session-affinity-mode: SOURCE_IP    # Enable the sticky session based on the source IP address.
```

Step 3 Use a browser to check whether the Service is available.

----End

Updating the Storage Class

As the storage infrastructures of clusters may be different, storage volumes cannot be mounted to the target cluster. You can use either of the following methods to update the volumes:

NOTICE

Both update methods can be performed only before the application is restored in the target cluster. Otherwise, PV data resources may fail to be restored. In this case, use Velero to restore applications after the storage class update is complete. For details, see**Restoring Applications in the Target Cluster**.

Method 1: Creating a ConfigMap mapping

Step 1 Create a ConfigMap in the CCE cluster and map the storage class used by the source cluster to the default storage class of the CCE cluster.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: change-storageclass-plugin-config
  namespace: velero
  labels:
    app.kubernetes.io/name: velero
    velero.io/plugin-config: "true"
    velero.io/change-storage-class: RestoreItemAction
data:
  {Storage class name01 in the source cluster}: {Storage class name01 in the target cluster}
  {Storage class name02 in the source cluster}: {Storage class name02 in the target cluster}
```

Step 2 Run the following command to apply the ConfigMap configuration:

```
$ kubectl create -f change-storage-class.yaml
configmap/change-storageclass-plugin-config created
```

----End

Method 2: Creating a storage class with the same name

Step 1 Run the following command to query the default storage class supported by CCE:

```
kubectl get sc
```

Information similar to the following is displayed:


```
NAME                PROVISIONER                  RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
csi-disk            everest-csi-provisioner      Delete          Immediate           true                   3d23h
csi-disk-topology   everest-csi-provisioner      Delete          WaitForFirstConsumer   true                3d23h
csi-nas             everest-csi-provisioner      Delete          Immediate           true                   3d23h
csi-obs             everest-csi-provisioner      Delete          Immediate           false                  3d23h
csi-sfsturbo        everest-csi-provisioner      Delete          Immediate           true                   3d23h
```

Table 3-6 Storage classes

Storage Class	Storage Resource
csi-disk	EVS
csi-disk-topology	EVS with delayed binding
csi-nas	SFS
csi-obs	OBS
csi-sfsturbo	SFS Turbo

Step 2 Run the following command to export the required storage class details in YAML format:

```
kubectl get sc <storageclass-name> -o=yaml
```

Step 3 Copy the YAML file and create a new storage class.

Change the storage class name to the name used in the source cluster to call basic storage resources of the cloud.

The YAML file of csi-obs is used as an example. Delete the unnecessary information in italic under the **metadata** field and modify the information in bold. You are advised not to modify other parameters.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  creationTimestamp: "2021-10-18T06:41:36Z"
  name: <your_storageclass_name>     # Use the name of the storage class used in the source cluster.
  resourceVersion: "747"
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-obs
  uid: 4dbbe557-ddd1-4ce8-bb7b-7fa15459aac7
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: obsfs
  everest.io/obs-volume-type: STANDARD
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

NOTE

- SFS Turbo file systems cannot be directly created using StorageClass. Go to the SFS Turbo console to create SFS Turbo file systems that belong to the same VPC subnet and have inbound ports (111, 445, 2049, 2051, 2052, and 20048) enabled in the security group.
- CCE does not support EVS disks of the ReadWriteMany type. If resources of this type exist in the source cluster, change the storage type to **ReadWriteOnce**.

Step 4 Restore the cluster application by referring to **Restoring Applications in the Target Cluster** and check whether the PVC is successfully created.


```
kubectl get pvc
```

In the command output, the **VOLUME** column indicates the name of the PV automatically created using the storage class.

```
NAME   STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc    Bound    pvc-4c8e655a-1dbc-4897-ae6c-446b502f5e77   5Gi        RWX            local          13s
```

----End

Updating Databases

In this example, the database is a local MySQL database and does not need to be reconfigured after the migration. If you use **DRS** to migrate a local database to RDS, configure database access based on site requirements after the migration.

NOTE

- If the RDS instance is in the same VPC as the CCE cluster, it can be accessed using the private IP address. Otherwise, it can only be accessed only through public networks by binding an EIP. You are advised to use the private network access mode for high security and good RDS performance.
- Ensure that the inbound rule of the security group to which RDS belongs has been enabled for the cluster. Otherwise, the connection will fail.

Step 1 Log in to the RDS console and obtain the private IP address and port number of the DB instance on the **Basic Information** page.

Step 2 Run the following command to modify the WordPress workload:

```
kubectl edit deploy wordpress
```

Set the environment variables in the **env** field.

- **WORDPRESS_DB_HOST**: address and port number used for accessing the database, that is, the internal network address and port number obtained in the previous step.
- **WORDPRESS_DB_USER**: username for accessing the database.
- **WORDPRESS_DB_PASSWORD**: password for accessing the database.
- **WORDPRESS_DB_NAME**: name of the database to be connected.

Step 3 Check whether the RDS database is properly connected.

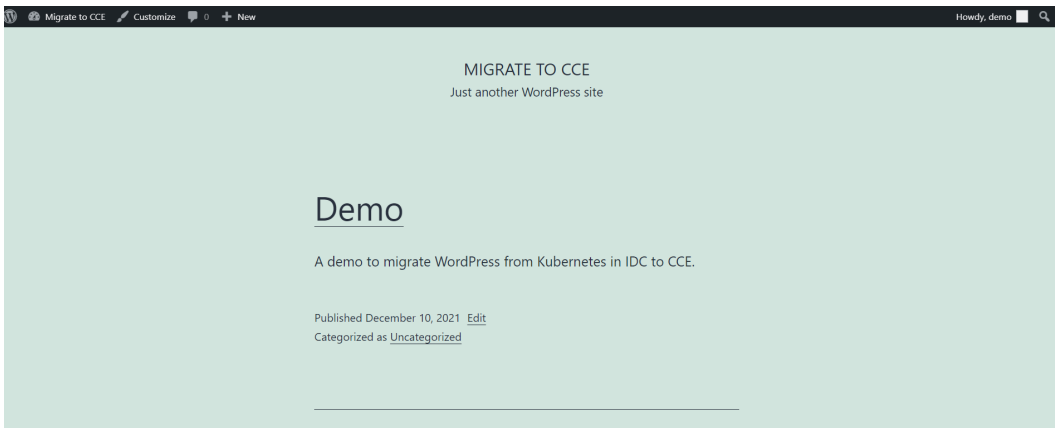
----End

3.2.3.5 Performing Additional Tasks

Verifying Application Functions

Cluster migration involves full migration of application data, which may cause intra-application adaptation problems. In this example, after the cluster is migrated, the redirection link of the article published in WordPress is still the original domain name. If you click the article title, you will be redirected to the application in the source cluster. Therefore, search for the original domain name in WordPress and replace it with the new domain name, change the values of **site_url** and primary URL in the database. For details, see **Changing The Site URL**.

Access the new address of the WordPress application. If the article published before the migration is displayed, the data of the persistent volume is successfully restored.



Switching Live Traffic to the Target Cluster

O&M personnel switch DNS to direct live traffic to the target cluster.

- DNS traffic switching: Adjust the DNS configuration to switch traffic.
- Client traffic switching: Upgrade the client code or update the configuration to switch traffic.

Bringing the Source Cluster Offline

After confirming that the service on the target cluster is normal, bring the source cluster offline and delete the backup files.

- Verify that the service on the target cluster is running properly.
- Bring the source cluster offline.
- Delete backup files.

3.2.3.6 Troubleshooting

Storage Volumes of the HostPath Type Cannot Be Backed Up

Both HostPath and Local volumes are local storage volumes. However, the Restic tool integrated in Velero cannot back up the PVs of the HostPath type and supports only the Local type. Therefore, you need to replace the storage volumes of the HostPath type with the Local type in the source cluster.

NOTE

It is recommended that Local volumes be used in Kubernetes v1.10 or later and can only be statically created. For details, see **local**.

Step 1 Create a storage class for the Local volume.

Example YAML:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
```



```
  name: local
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Step 2 Change the **hostPath** field to the **local** field, specify the original local disk path of the host machine, and add the **nodeAffinity** field.

Example YAML:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    app: mysql
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 5Gi
  storageClassName: local     # Storage class created in the previous step
  persistentVolumeReclaimPolicy: Delete
  local:
    path: "/mnt/data"     # Path of the attached local disk
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: Exists
```

Step 3 Run the following commands to verify the creation result:

```
kubectl get pv
```

Information similar to the following is displayed:

```
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM             STORAGECLASS
REASON   AGE
mysql-pv   5Gi        RWO            Delete           Available                     local
 3s
```

----End

Backup Tool Resources Are Insufficient

In the production environment, if there are many backup resources, for example, the default resource size of the backup tool is used, the resources may be insufficient. In this case, perform the following steps to adjust the CPU and memory size allocated to the Velero and Restic:

Before installing Velero:

You can specify the size of resources used by Velero and Restic when **installing Velero**.

The following is an example of installation parameters:

```
velero install \
  --velero-pod-cpu-request 500m \
  --velero-pod-mem-request 1Gi \
  --velero-pod-cpu-limit 1000m \
  --velero-pod-mem-limit 1Gi \
  --use-node-agent \
  --node-agent-pod-cpu-request 500m \
  --node-agent-pod-mem-request 1Gi \
  --node-agent-pod-cpu-limit 1000m \
  --node-agent-pod-mem-limit 1Gi
```


After Velero is installed:

Step 1 Edit the YAML files of the Velero and node-agent workloads in the **velero** namespace.

```
kubectl edit deploy velero -n velero
kubectl edit ds node-agent -n velero
```

Step 2 Modify the resource size under the **resources** field. The modification is the same for the Velero and Restic workloads, as shown in the following:

```
resources:
  limits:
    cpu: "1"
    memory: 1Gi
  requests:
    cpu: 500m
    memory: 1Gi
```

----End

4 DevOps

4.1 Installing, Deploying, and Interconnecting Jenkins with SWR and CCE Clusters

4.1.1 Solution Overview

What Is Jenkins?

Jenkins is an open source continuous integration (CI) tool that provides user-friendly GUIs. It originates from Hudson and is used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Jenkins is written in Java and can run in popular servlet containers such as Tomcat, or run independently. It is usually used together with the version control tools (or SCM tools) and build tools. Jenkins supports various languages and is compatible with third-party build tools, such as Maven, Ant, and Gradle. It seamlessly integrates with common version control tools, such as SVN and Git, and can directly connect to source code hosting services, such as GitHub.

Constraints

- This solution can be deployed only in CCE clusters. It is not supported in DeC.
- CCE does not provide maintenance and support for Jenkins. The maintenance is provided by the developers.

Solution Architecture

You can install Jenkins using the following methods:

- You can use a single Master to install Jenkins. The Master handles jobs and builds and releases services. However, security risks may exist.
- Another one is to use Master+Agents. Master schedules build jobs to Agents for execution, and monitors Agent status. Agents execute build jobs dispatched by the Master and return the job progress and result.

You can install the Master and Agents on VMs, containers, or combination of the two. For details, see **Table 4-1**.

Table 4-1 Jenkins deployment modes

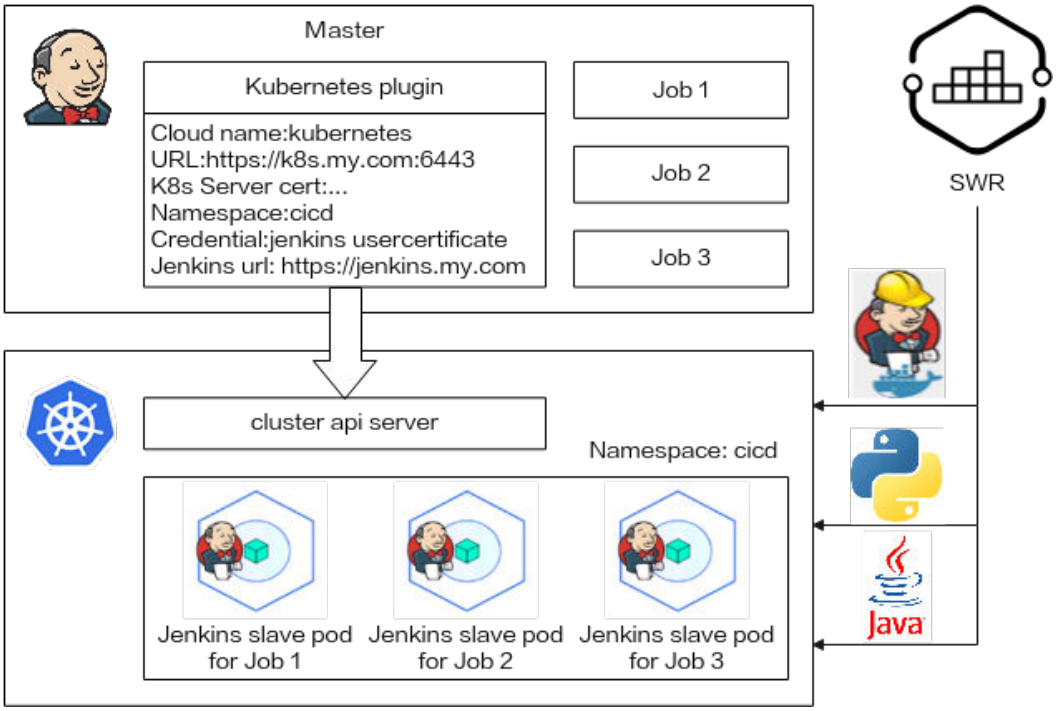
Deployment Mode	Master	Agent	Advantages and Disadvantages
Single Master	VMs	-	• Advantage: Localized construction is easy to operate. • Disadvantage: Job management and execution are performed on the same VM and the security risk is high.
Single Master	Containers	-	• Advantage: Kubernetes containers support self-healing. • Disadvantage: Job management and execution are not isolated. Security risks exist.
Master+Agents	VMs	VMs	• Advantage: Job management and execution are isolated and the security risk is low. • Disadvantage: Agents are fixed. Resources cannot be scheduled and the resource utilization is low and the maintenance cost is high.
		Containers (Kubernetes cluster)	• Advantage: Containerized Agents can be fixed or dynamic. Kubernetes schedules the dynamic Agents, improving the resource utilization. Jobs can be evenly allocated based on the scheduling policy, which is easy to maintain. • Disadvantage: The Master may break down and the recovery cost is high.

Deployment Mode	Master	Agent	Advantages and Disadvantages
Master+Agents	Containers (Kubernetes cluster)	Containers (Kubernetes cluster)	• Advantage: Containerized Agents can be fixed or dynamic. Kubernetes schedules the dynamic Agents, improving the resource utilization. The Master is self-healing and the maintenance cost is low. Agents and the Master can be deployed in the same cluster or in different clusters. • Disadvantage: The system is complex and the environment is difficult to set up.

In this section, Jenkins is installed with the containerized Master and Agents. Kubernetes schedules the dynamic Agents. For details about the architecture, see **Figure 4-1**.

- The Master handles jobs. Install Kubernetes add-ons on the Master to use the Kubernetes platform resources.
- The Kubernetes platform generates pods for Agents to execute jobs. When a job is scheduled on the Master, the Master sends a request to the Kubernetes platform using the Kubernetes add-on. After receiving the request, Kubernetes builds a pod using the pod template to send requests to the Master. After the Master is successfully connected, you can execute the job on the pod.

Figure 4-1 Installing Jenkins on Kubernetes



Procedure

Step 1 **Installing and Deploying Jenkins Master**

Jenkins Master is deployed in the CCE cluster using container images.

Step 2 **Configuring Jenkins Agent**

Jenkins can fix Agents in the cluster or use the pipeline to interconnect with CCE to provide pods for Agents to execute jobs. The dynamic Agents use Kubernetes add-ons to configure cluster authentication and user permissions.

Step 3 **Using Jenkins to Build a Pipeline**

The Jenkins pipeline interconnects with SWR and calls **docker build/login/push** commands in Agents to package and push images automatically.

You can also use pipelines to deploy and upgrade Kubernetes resources (such as Deployments, Services, ingresses, and jobs).

----End

4.1.2 Resource and Cost Planning

NOTICE

The fees listed here are estimates. The actual fees will be displayed on the Huawei Cloud console.

The required resources are as follows:

Table 4-2 Resource and cost planning

Resource	Description	Quantity	Estimated Fee (USD)
Cloud Container Engine (CCE)	Pay-per-use recommended • Cluster type: CCE cluster • CCE cluster version: v1.25 • Cluster scale: 50 nodes • HA: Yes	1	2.91/hour
VM	Pay-per-use recommended • VM type: General computing-plus • Specifications: 4 vCPUs | 8 GiB • OS: EulerOS 2.9 • System disk: 50 GiB | General-purpose SSD • Data disk: 100 GiB | General-purpose SSD	1	1.00/hour
Elastic Volume Service (EVS)	Pay-per-use recommended • EVS disk specifications: 100 GiB • EVS disk type: General-purpose SSD	1	0.1/hour
Load Balancer (ELB)	Pay-per-use recommended • Type: Shared • Billed By: Traffic • Bandwidth: 5 Mbit/s	1	0.32/hour + 0.80/GiB (The traffic fee is charged based on the actual outbound traffic.)

4.1.3 Procedure

4.1.3.1 Installing and Deploying Jenkins Master

NOTE

On the Jenkins page, the UI strings in Chinese and English are different. The screenshots in this section are for your reference only.

Selecting an Image

Select a relatively new, stable image from Docker Hub. For this test, select **jenkinsci/blueocean**, which is bound with all Blue Ocean add-ons and functions. There is no need to install Blue Ocean add-ons separately. For details, see **Installing Jenkins**.

Preparations

- Before creating a containerized workload, buy a cluster (the cluster must contain at least one node with four vCPUs and 8 GiB memory). For details, see **Buying a CCE Cluster**.

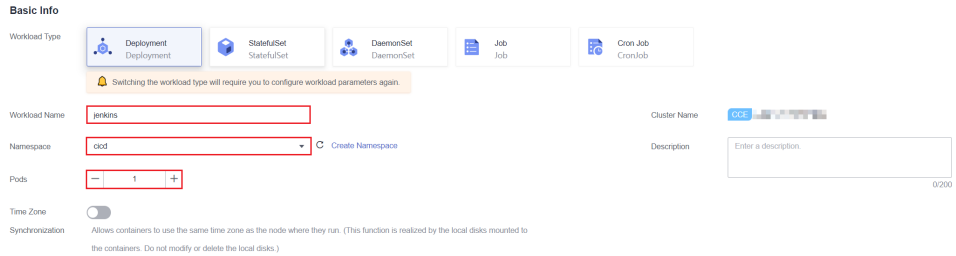
  The Docker in Docker scenario is required, which is, running the Docker commands in the container. Select the Docker container engine for the node.
- To enable external networks to access the workload, ensure that an elastic IP address (EIP) has been bound to or a load balancer has been configured for at least one node in the cluster.

Installing and Deploying Jenkins on CCE

Step 1 Log in to the CCE console, choose **Workloads** > **Deployments** and click **Create Workload** on the upper right corner.

Step 2 Configure basic workload parameters.

- **Workload Name**: jenkins (customizable)
- **Namespace**: Select the namespace where Jenkins will be deployed. You can create a namespace.
- **Pods**: Set it to **1**.

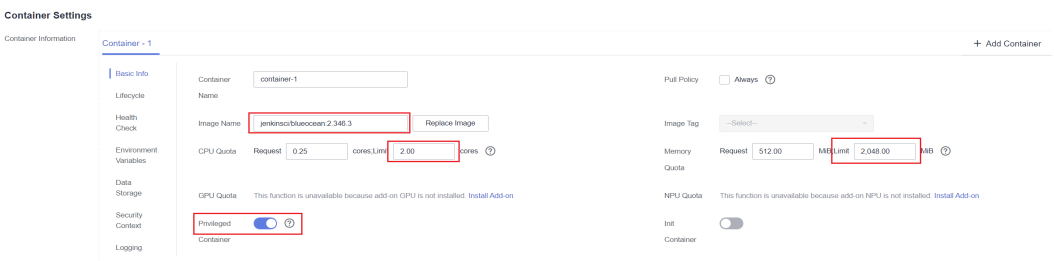


Step 3 Configure basic container parameters.

- **Image Name**: Enter **jenkinsci/blueocean**. Select an image version as required. If no version is selected, the latest version will be used by default.
- **CPU Quota**: Set **Limit** to **2** cores.
- **Memory Quota**: Set **Limit** to **2048** MiB.
- **Privileged Container**: If Jenkins is deployed with a single Master, enable **Privileged Container** so that the container can perform operations on the host. Otherwise, Docker commands cannot be executed in the Jenkins Master container.

Retain the default values for other parameters.

Figure 4-2 Basic container parameters



Step 4 Choose **Data Storage** > **PersistentVolumeClaims (PVCs)** and add a persistent volume.

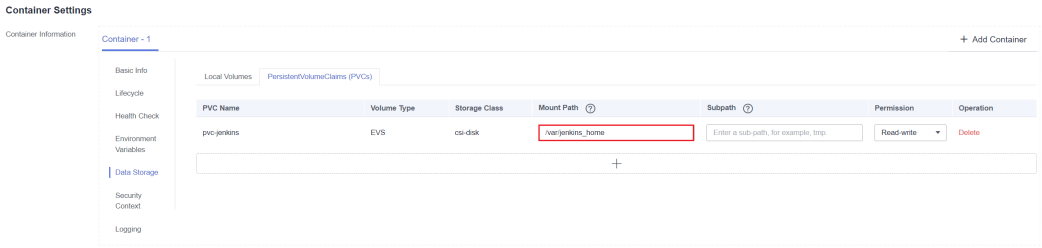
In the displayed dialog box, select a cloud volume and enter **/var/jenkins_home** in the mount path to mount a cloud volume for Jenkins to store data persistently.

NOTE

The cloud storage type can be **EVS** or **SFS**. If no cloud storage is available, click **Create PVC**.

If you select **EVS**, the AZ of the EVS disk must be the same as that of the node.

Figure 4-3 Adding a cloud volume



Step 5 Add permissions to the Jenkins container so that related commands can be executed in it.

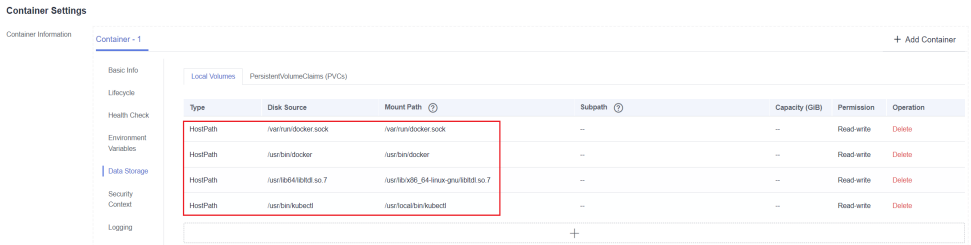
1. Ensure that **Privileged Container** is enabled in **3**.
2. Choose **Data Storage** > **Local Volumes**, add a local volume, and mount the host path to the corresponding container path.

Table 4-3 Mounting path

Storage Type	Host Path	Mounting Path
hostPath	**/var/run/docker.sock**	**/var/run/docker.sock**
hostPath	**/usr/bin/docker**	**/usr/bin/docker**
hostPath	**/usr/lib64/libltdl.so.7**	**/usr/lib/x86_64-linux-gnu/libltdl.so.7**
hostPath	**/usr/bin/kubectl**	**/usr/local/bin/kubectl**

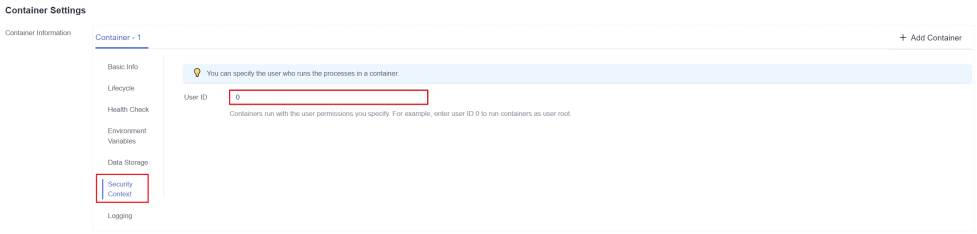
After the mounting is complete, the page shown in **Figure 4-4** is displayed.

Figure 4-4 Mounting the host paths to the corresponding container paths



3. In **Security Context**, set **User ID** to **0** (user **root**).

Figure 4-5 Configuring the user



Step 6 Specify the access mode in **Service Configuration**.

The Jenkins container image has two ports: 8080 and 50000. Configure them separately. Port 8080 is used for web login, and port 50000 is used for the connection between Master and Agent.

In this example, two Services are created:

- **LoadBalancer**: provides external web access using port 8080. You can also select **NodePort** to provide external access.

  Set the Service name to **jenkins** (customizable), the container port to **8080**, the access port to **8080**, and retain the default values for other parameters.

- **ClusterIP**: used by the Agent to connect to the Master. The IP addresses of **jenkins-web** and **jenkins-agent** need to be the same. Therefore, port 8080 for web access and port 50000 for agent access are included.

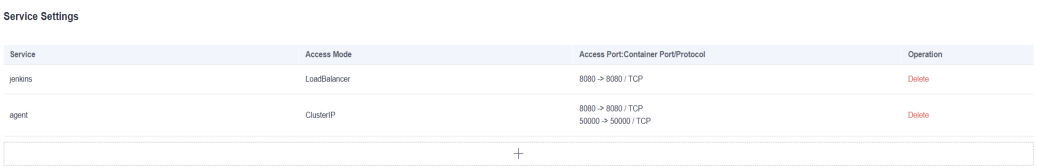
  Set the Service name to **agent** (customizable), the container port 1 to **8080**, the access port 1 to **8080**, the container port 2 to **50000**, the access port 2 to **50000**, and retain the default values for other parameters.

  NOTE

  In this example, Agents and the Master are deployed in the same cluster. Therefore, the Agents can use the ClusterIP Service to connect to the Master.

  If Agents need to connect to the Master across clusters or through the public network, select a proper Service type. Note that the IP addresses of **Jenkins-web** and **Jenkins-agent** need to be the same. Therefore, **ports 8080 and 50000 must be enabled for the IP address connected to jenkins-agent**. For addresses used only for web access, enable only the port 8080.

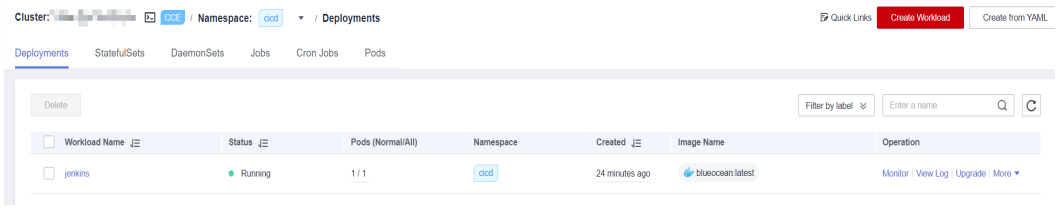
Figure 4-6 Adding a Service



Step 7 Retain the default settings for **Advanced Settings** and click **Create Workload**.

Step 8 Click **Back to Deployment List** to view the Deployment status. If the workload is in the **Running** status, the Jenkins application is accessible.

Figure 4-7 Viewing the workload status

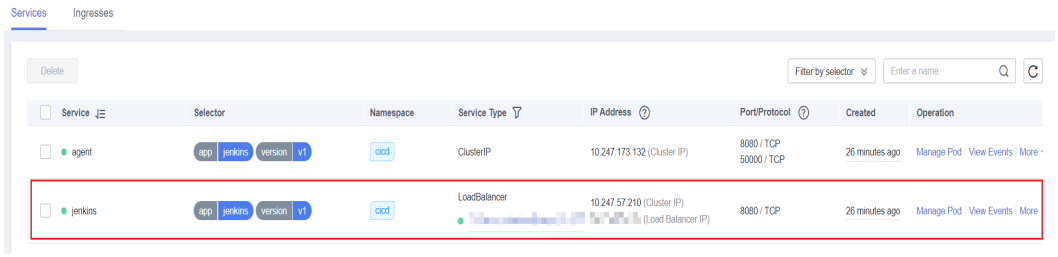


----End

Logging In and Initializing Jenkins

Step 1 On the CCE console, click the target cluster. Choose **Networking** in the navigation pane. On the **Services** tab page, view the Jenkins access mode.

Figure 4-8 Access mode corresponding to port 8080

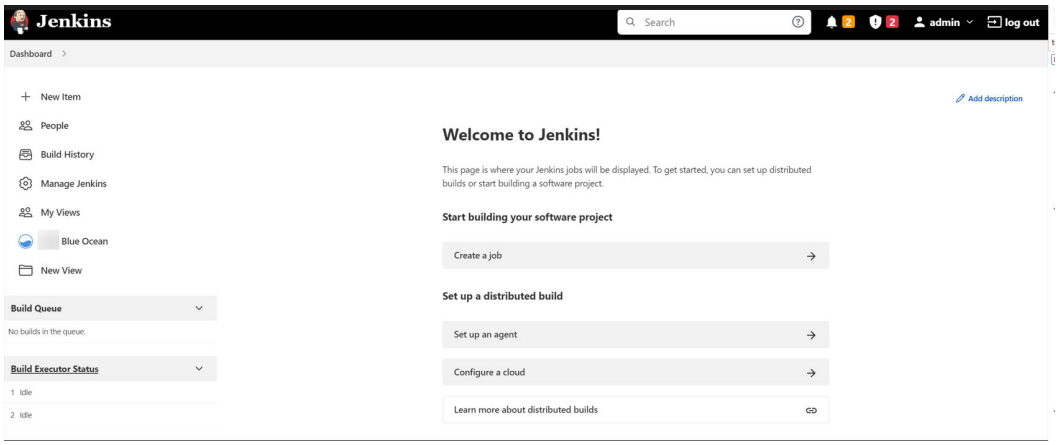


Step 2 Enter **EIP:8080** of the load balancer in the browser address box to visit the Jenkins configuration page.

When you visit the page for the first time, you are prompted to obtain the initial administrator password. You can obtain the password from the Jenkins pod. Before running the following commands, connect to the cluster using kubectl. For details, see **Connecting to a Cluster Using kubectl**.

```
# kubectl get pod -n cicd
NAME                          READY   STATUS    RESTARTS   AGE
jenkins-7c69b6947c-5gvlm      1/1     Running   0          17m
# kubectl exec -it jenkins-7c69b6947c-5gvlm -n cicd -- /bin/sh
# cat /var/jenkins_home/secrets/initialAdminPassword
b10eabe29a9f427c9b54c01a9c3383ae
```

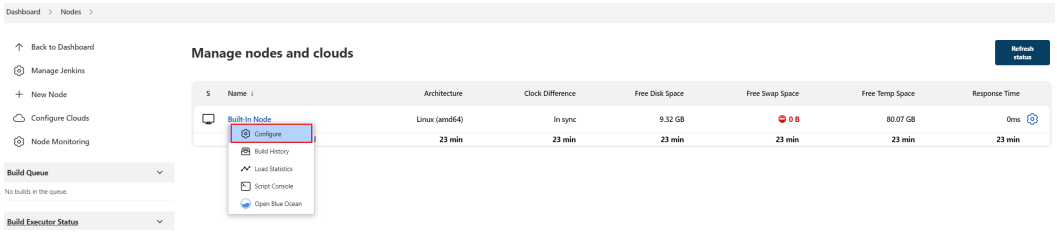
Step 3 The system prompts you to select the default recommended add-on and create an administrator upon the first login. After the initial configuration is complete, the Jenkins page is displayed.



----End

Modifying the Number of Concurrent Build Jobs

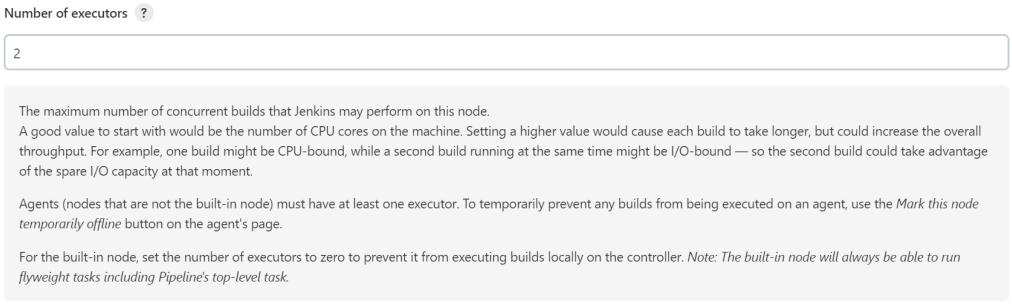
Step 1 On the Jenkins dashboard page, click **Manage Jenkins** on the left, choose **System Configuration** > **Manage nodes and clouds**, and select **Configure** from the drop-down list of the target node.



NOTE

- You can modify the number of concurrent build jobs on both Master and Agent. The following uses Master as an example.
- If the **Master is used with Agents**, you are advised to set the number of concurrent build jobs of Master to **0**. That is, all build jobs are performed using Agents. If a **single Master** is used, you do not need to change the value to **0**.

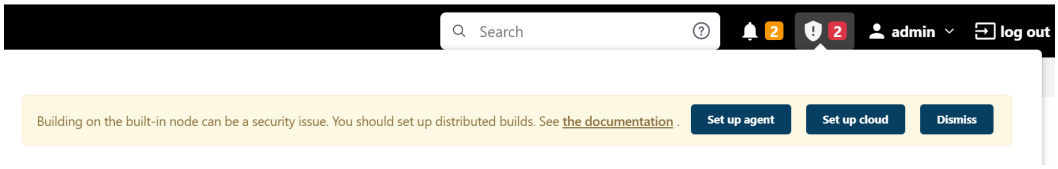
Step 2 Modify the maximum number of concurrent build jobs. In this example, the value is changed to **2**. You can change the value as required.



----End

4.1.3.2 Configuring Jenkins Agent

After Jenkins is installed, the following information may display, indicating that Jenkins uses a Master for local build and Agents are not configured.



If you install Jenkins using a Master, you can build a pipeline after performing operations in **Installing and Deploying Jenkins Master**. For details, see **Using Jenkins to Build a Pipeline**.

If you install Jenkins using a Master and Agents, you can select either of the following solutions to configure Agents.

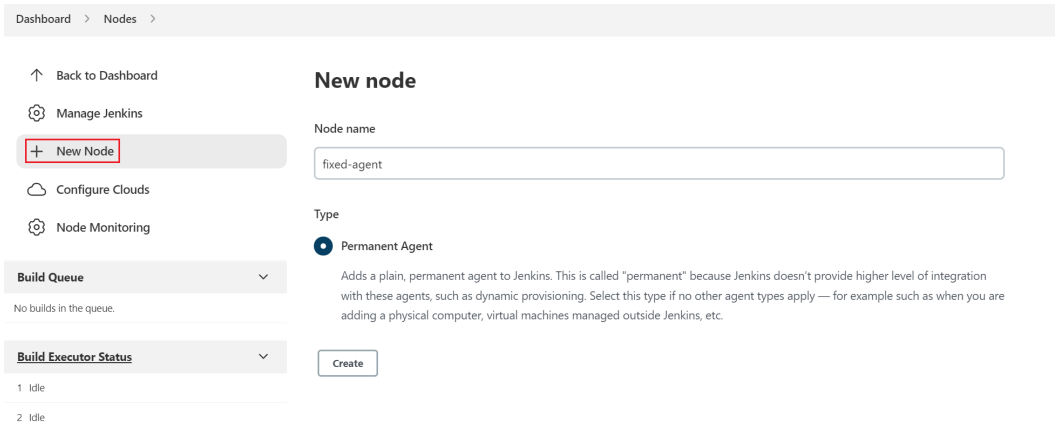
- **Fixed Agent**: The Agent container keeps running and occupying cluster resources after a job is built. This configuration is simple.
- **Dynamic Agent**: An Agent container is dynamically created during job build and is killed after the job is built. In this way, resources can be dynamically allocated and the resource utilization is high. This configuration is complex.

In this section, the Agent is containerized using the **jenkins/inbound-agent:4.13.3-1** image.

Adding a Fixed Agent to Jenkins

Step 1 Log in to the Jenkins dashboard, click **Manage Jenkins** on the left, and choose **System Configuration** > **Manage nodes and clouds**.

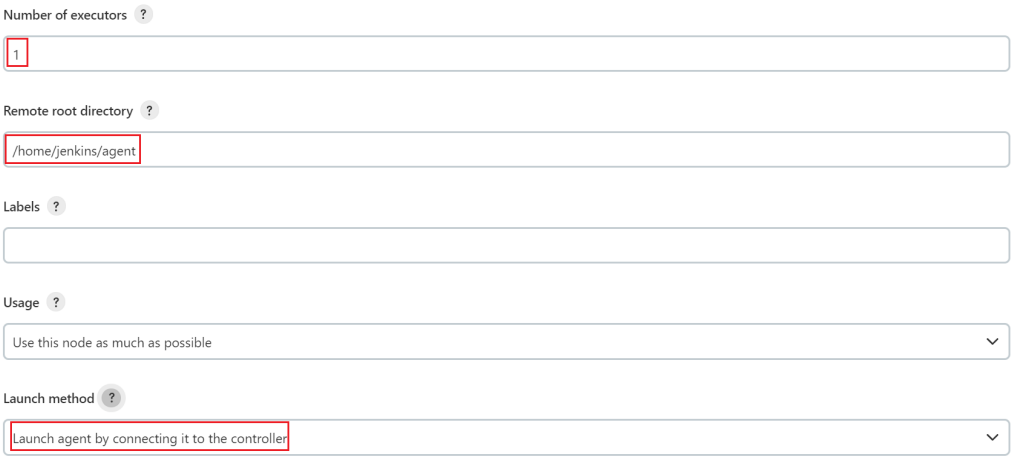
Step 2 Click **New Node** on the left, enter the node name **fixed-agent** (which can be customized), and select **Permanent Agent** for **Type**.



Step 3 Specify the following node information:

- **Number of executors**: The default value is **1**. Set this parameter as required.
- **Remote root directory**: Enter **/home/jenkins/agent**.
- Launch method: Select **Launch agent by connecting it to the controller**.

Retain the values for other parameters and click **Save**.

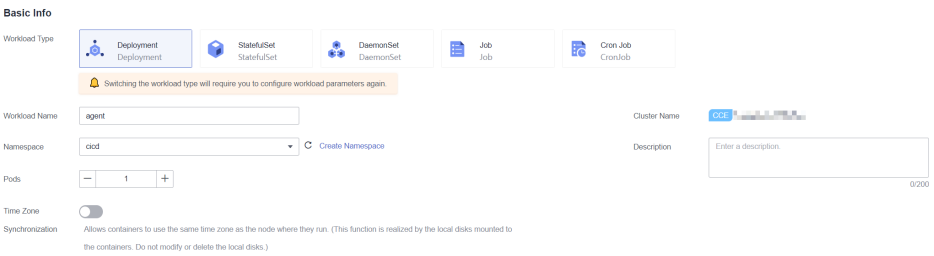


Step 4 In the **Nodes** page, click the new node. The Agent status is disconnected, and the method for connecting the node to Jenkins is provided. This command applies to VM installation. In this example, container-based installation is used. Therefore, you only need to copy the secret, as shown in the following figure.

Step 5 Log in to the CCE console, click the target cluster. Choose **Workloads** > **Deployments** and click **Create Workload** on the right.

Step 6 Configure basic workload parameters.

- **Workload Name**: agent (user-defined)
- **Namespace**: Select the namespace where Jenkins will be deployed. You can create a namespace.
- **Pods**: Set it to **1**.

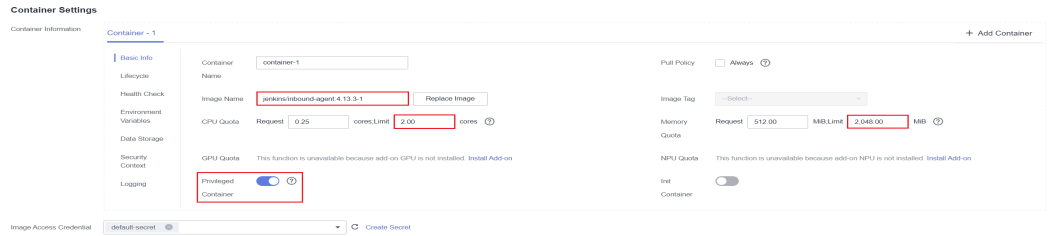


Step 7 Configure basic container parameters.

- **Image Name**: Enter **jenkins/inbound-agent:4.13.3-1**. The image version may change with time. Select an image version as required or use the latest version.

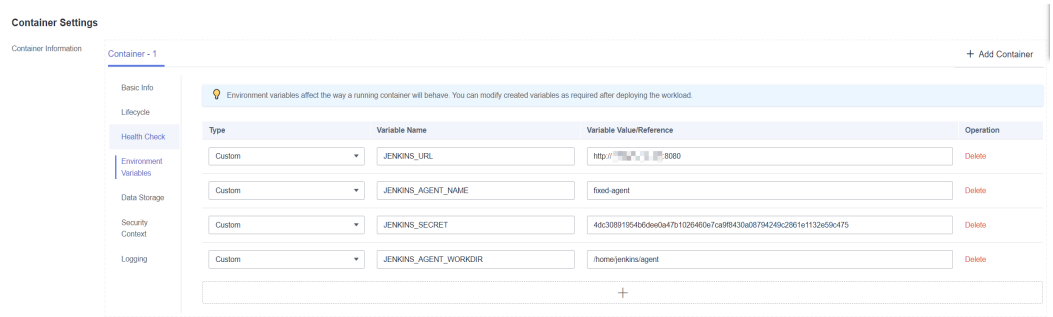
- **CPU Quota**: In this example, set **Limit** to **2** cores.
- **Memory Quota**: Set **Limit** to **2048** MiB.
- **Privileged Container**: must be enabled so that the container can obtain permissions on the host. Otherwise, Docker commands cannot be executed in the container.

Retain the default values for other parameters.



Step 8 Run the following commands to configure the environment variables:

- **JENKINS_URL**: access path of Jenkins. Enter the IP address of port 8080 set in **Step 6** (ports 8080 and 50000 must be enabled for the IP address), for example, **http://10.247.222.254:8080**.
- **JENKINS_AGENT_NAME**: name of the Agent set in **Step 2**. In this example, the value is **fixed-agent**.
- **JENKINS_SECRET**: secret copied from **Step 4**.
- **JENKINS_AGENT_WORKDIR**: remote work directory configured in **Step 3**, that is, **/home/jenkins/agent**.



Step 9 Add permissions to the Jenkins container so that Docker commands can be executed in the Jenkins container.

1. Ensure that **Privileged Container** is enabled in **3**.
2. Choose **Data Storage** > **Local Volumes**, add a local volume, and mount the host path to the corresponding container path.

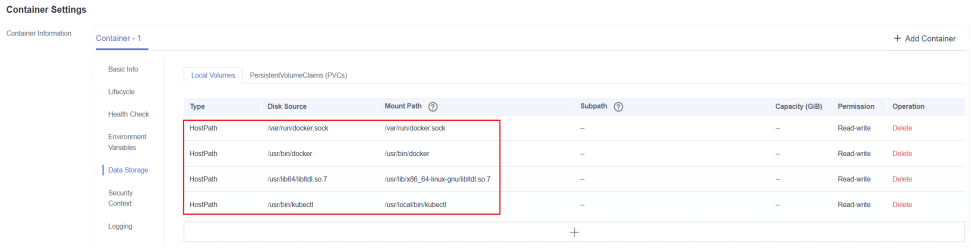
Table 4-4 Mounting path

Storage Type	Host Path	Mounting Path
hostPath	**/var/run/docker.sock**	**/var/run/docker.sock**
hostPath	**/usr/bin/docker**	**/usr/bin/docker**

Storage Type	Host Path	Mounting Path
hostPath	**/usr/lib64/libltdl.so.7**	**/usr/lib/x86_64-linux-gnu/libltdl.so.7**
hostPath	**/usr/bin/kubectl**	**/usr/local/bin/kubectl**

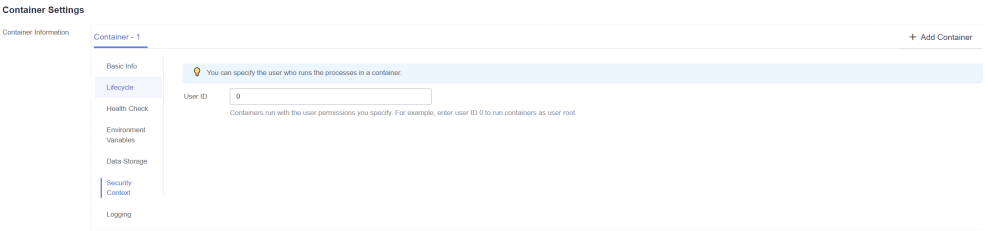
After the mounting is complete, the page shown in **Figure 4-9** is displayed.

Figure 4-9 Mounting the host paths to the corresponding container paths



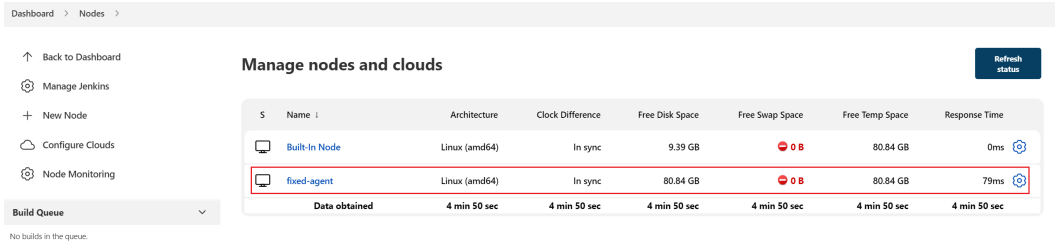
3. In **Security Context**, set **User ID** to **0** (user **root**).

Figure 4-10 Configuring the user



Step 10 Retain the default settings for **Advanced Settings** and click **Create Workload**.

Step 11 Go to the Jenkins page and refresh the node status to **In sync**.



NOTE

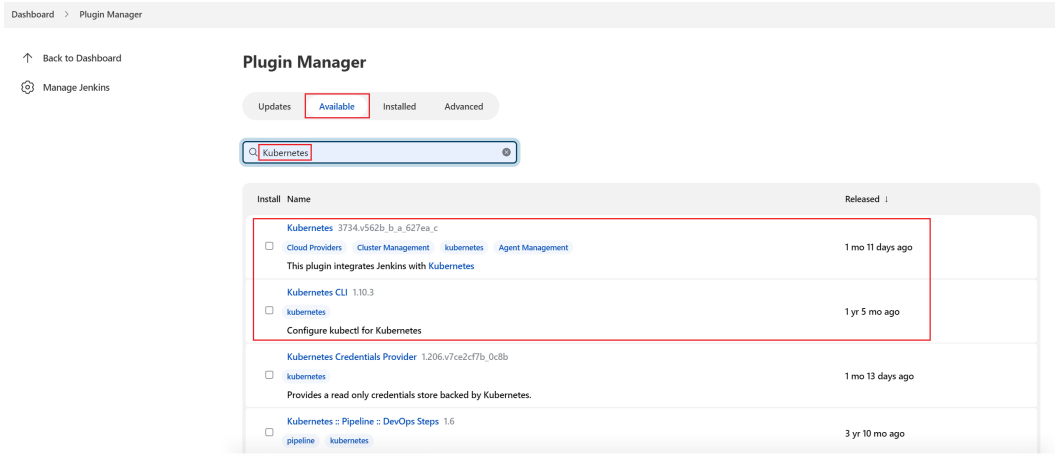
After the Agent is configured, you are advised to set the number of concurrent build jobs of the Master to **0**. That is, you use the Agent for build. For details, see **Modifying the Number of Concurrent Build Jobs**.

----End

Setting a Dynamic Agent for Jenkins

Step 1 Install the plug-in.

On the Jenkins dashboard page, click **Manage Jenkins** on the left and choose **System Configuration** > **Manage Plugins**. On the **Available** tab, search for **Kubernetes** and install **Kubernetes CLI** and **Kubernetes**.



The plug-in version may change with time. Select a plug-in version as required.

- **Kubernetes Plugin**: 3734.v562b_b_a_627ea_c

  It is used to run dynamic Agents in the Kubernetes cluster, create a Kubernetes pod for each started Agent, and stop the pod after each build is complete.

- **Kubernetes CLI Plugin**: 1.10.3

  kubectl can be configured for jobs to interact with Kubernetes clusters.

NOTE

The Jenkins plug-ins are provided by the plug-in maintainer and may be iterated due to security risks.

Step 2 Add cluster access credentials to Jenkins.

Add cluster access credentials to Jenkins in advance. For details, see **Setting Cluster Access Credentials**.

Step 3 Configure basic cluster information.

On the Jenkins dashboard page, click **Manage Jenkins** on the left and choose **System Configuration** > **Manage nodes and clouds**. Click **Configure Clouds** on the left to configure the cluster. Click **Add a new cloud** and select **Kubernetes**. The cluster name can be customized.

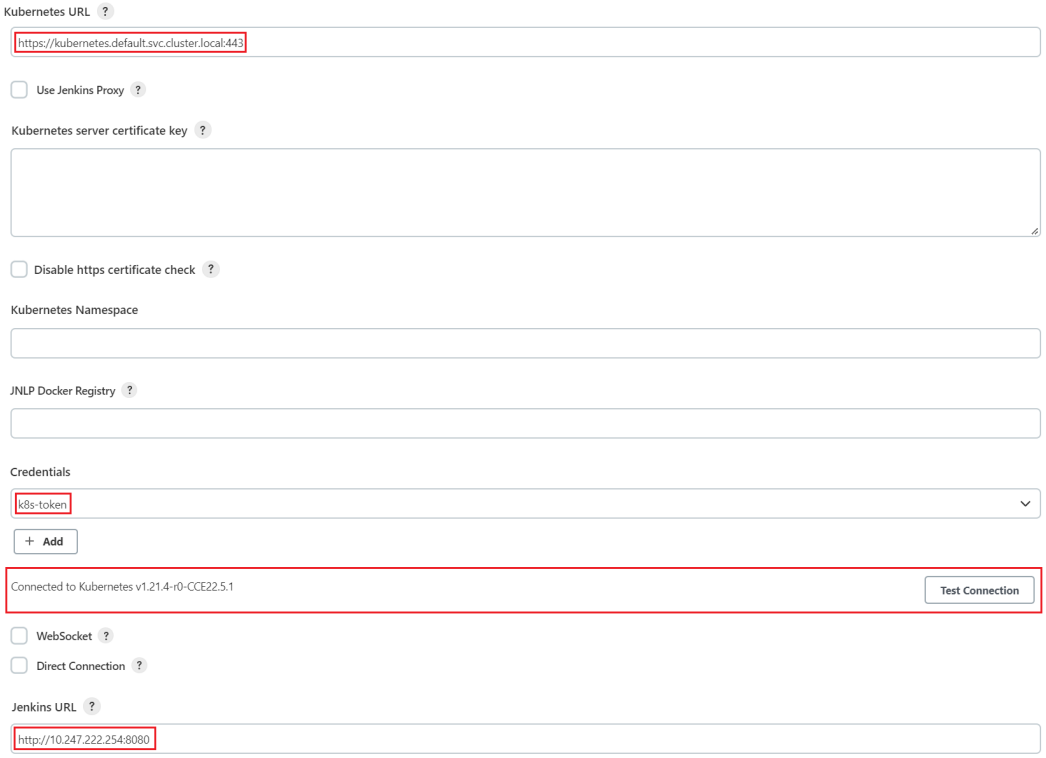
Step 4 Enter Kubernetes Cloud details.

Set the following cluster parameters and retain the values for other parameters, as shown in **Figure 4-11**.

- **Kubernetes URL**: cluster API server address. You can enter **https://kubernetes.default.svc.cluster.local:443**.
- **Credentials**: Select the cluster credential added in **Step 2**. You can click **Test Connection** to check whether the cluster is connected.

- **Jenkins URL**: Jenkins access path. Enter the IP address of port 8080 set in **Step 6** **(ports 8080 and 50000 must be enabled for the IP address, that is, the intra-cluster access address)**, for example, **http://10.247.222.254:8080**.

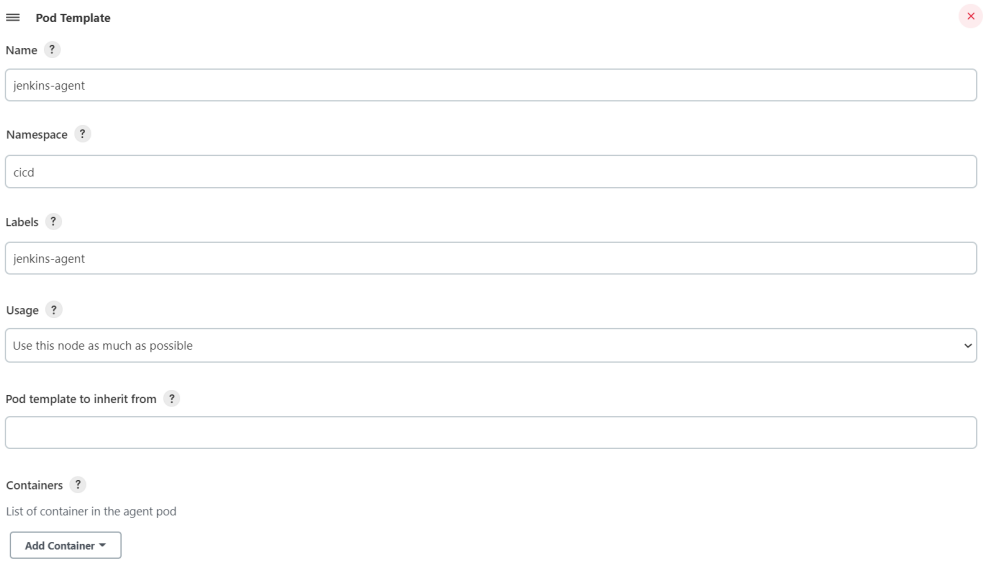
Figure 4-11 Example



Step 5 **Pod Template**: Click **Add Pod Template > Pod Template details** and set pod template parameters.

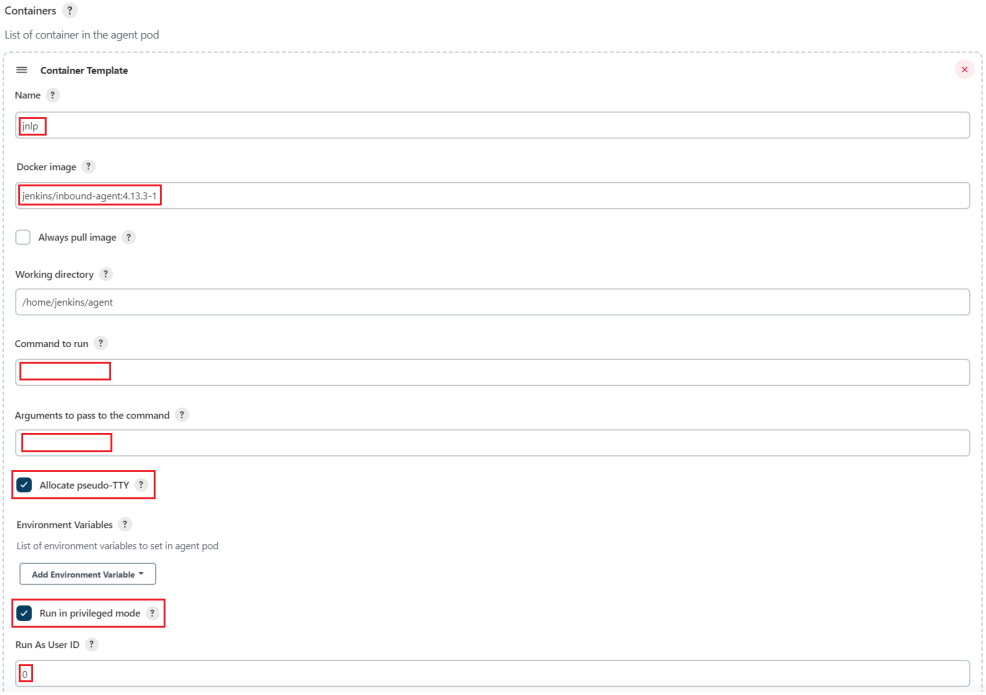
- Set the basic parameters of the pod template, as shown in **Figure 4-12**.
  - **Name**: **jenkins-agent**
  - **Namespace**: **cicd**
  - **Labels**: **jenkins-agent**
  - **Usage**: Select **Use this node as much as possible**.

Figure 4-12 Basic parameters of the pod template



- Add a container. Click **Add Container** > **Container Template**. **Figure 4-13** shows the parameters.
  - **Name**: The value must be **jnlp**.
  - **Docker image**: **jenkins/inbound-agent:4.13.3-1**. The image version may change with time. Select an image version as required or use the latest version.
  - **Working directory**: **/home/jenkins/agent** is selected by default.
  - **Command to run**/**Arguments to pass to the command**: Delete the existing default value and leave these two parameters empty.
  - **Allocate pseudo-TTY**: Select this parameter.
  - Select **Run in privileged mode** and set **Run As User ID** to **0** (**root** user).

Figure 4-13 Container template parameters



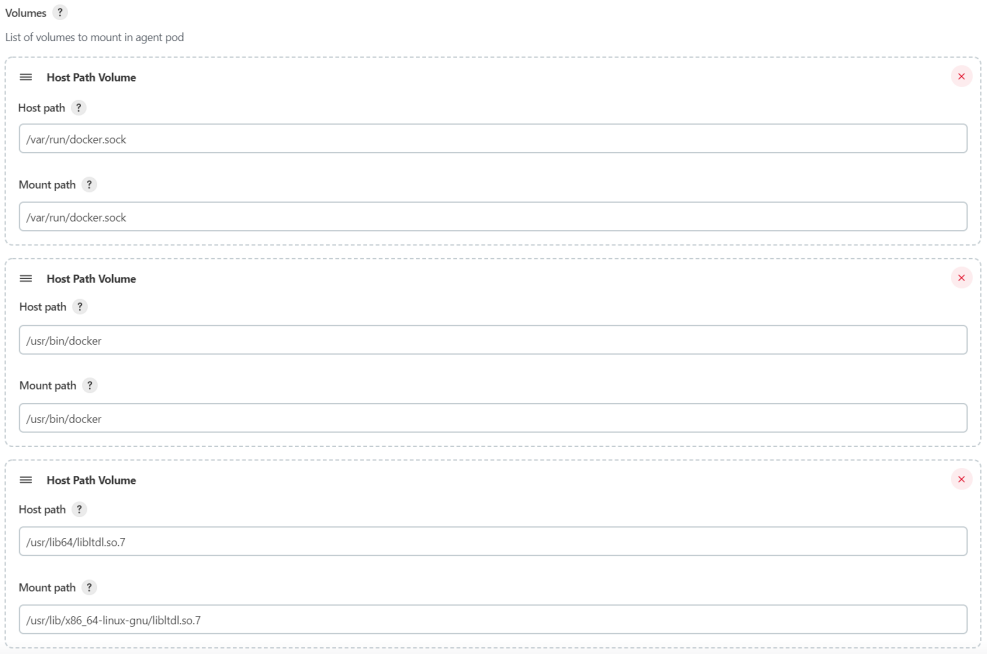
- Add a volume: Choose **Add Volume > Host Path Volume** to mount the host path in **Table 4-5** to the corresponding path of the container.

Table 4-5 Mounting path

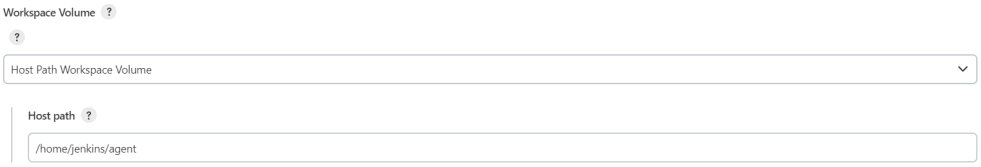
Storage Type	Host Path	Mounting Path
hostPath	**/var/run/docker.sock**	**/var/run/docker.sock**
hostPath	**/usr/bin/docker**	**/usr/bin/docker**
hostPath	**/usr/lib64/libltdl.so.7**	**/usr/lib/x86_64-linux-gnu/libltdl.so.7**
hostPath	**/usr/bin/kubectl**	**/usr/local/bin/kubectl**

After the mounting is complete, the page shown in **Figure 4-14** is displayed.

Figure 4-14 Mounting the host paths to the corresponding container paths



- **Run As User ID**: **0** (**root** user)
- **Workspace Volume**: working directory of the agent. Persistence is recommended. Select **Host Path Workspace Volume** and set **Host path** to **/home/jenkins/agent**.



Step 6 Click **Save**.

NOTE

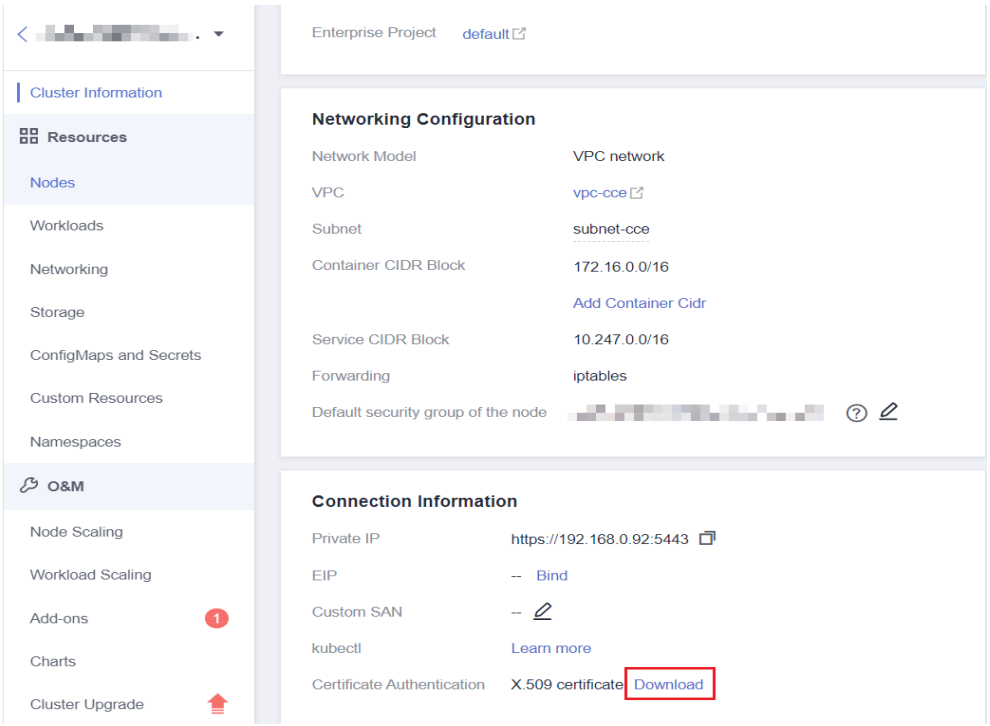
After the Agent is configured, you are advised to set the number of concurrent build jobs of the Master to **0**. That is, you use the Agent for build. For details, see **Modifying the Number of Concurrent Build Jobs**.

----End

Setting Cluster Access Credentials

The certificate file that can be identified in Jenkins is in PKCS#12 format. Therefore, convert the cluster certificate to a PFX certificate file in PKCS#12 format.

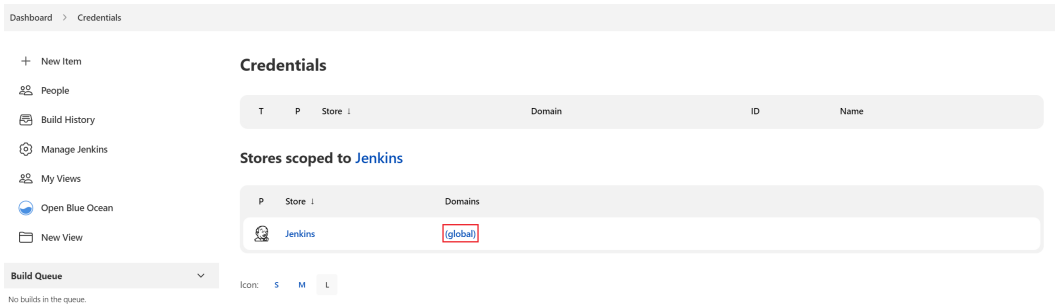
Step 1 Log in to the CCE console and go to the cluster console. Choose **Cluster Information > Connection Information** to download the cluster certificate. The downloaded certificate contains three files: **ca.crt**, **client.crt**, and **client.key**.



Step 2 Log in to a Linux host, place the three certificate files in the same directory, and use OpenSSL to convert the certificate into a **cert.pfx** certificate. After the certificate is generated, the system prompts you to enter a custom password.

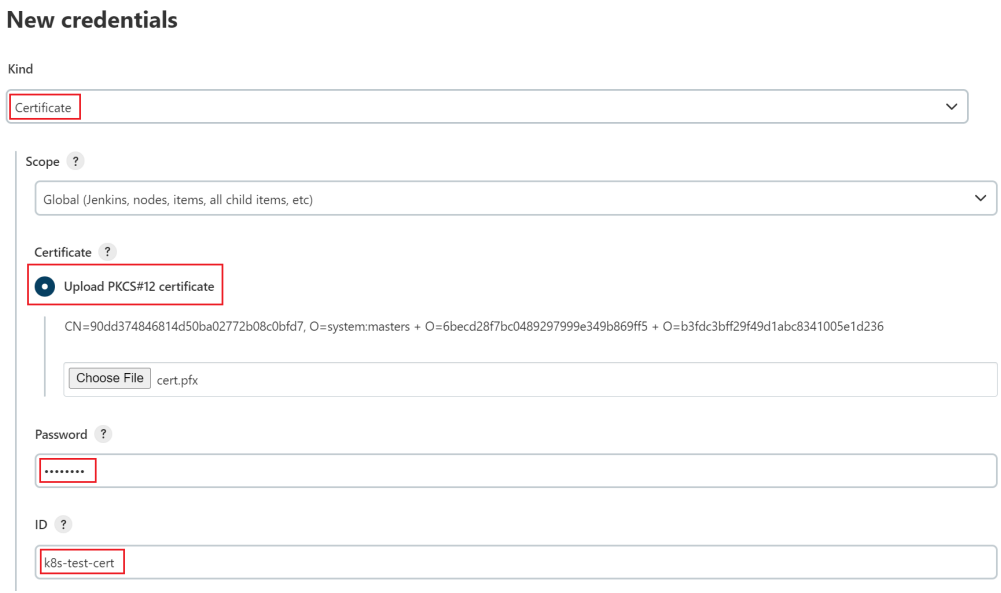
```
openssl pkcs12 -export -out cert.pfx -inkey client.key -in client.crt -certfile ca.crt
```

Step 3 On the Jenkins console, choose **Manage Jenkins** > **Manage Credentials** and click **Global**. You can also create a domain.



Step 4 Click **Add Credential**.

- **Kind**: Select **Certificate**.
- **Scope**: Select **Global**.
- **Certificate**: Select **Upload PKCS#12 certificate** and upload the **cert.pfx** file generated in **Step 2**.
- **Password**: The password customized during **cert.pfx** conversion.
- **ID**: Set this parameter to **k8s-test-cert**, which can be customized.



----End

4.1.3.3 Using Jenkins to Build a Pipeline

Obtaining a Long-Term Valid Login Command

During Jenkins installation and deployment, the Docker commands have been configured in the container (see **9**). Therefore, no additional configuration is required for interconnecting Jenkins with SWR. You can directly run the Docker commands. You only need to obtain a long-term valid SWR login command. For details, see **Obtaining a Long-Term Valid Login Command**.

For example, the command of this account is as follows:

```
docker login -u ap-southeast-1@xxxxx -p xxxxx swr.ap-southeast-1.myhuaweicloud.com
```

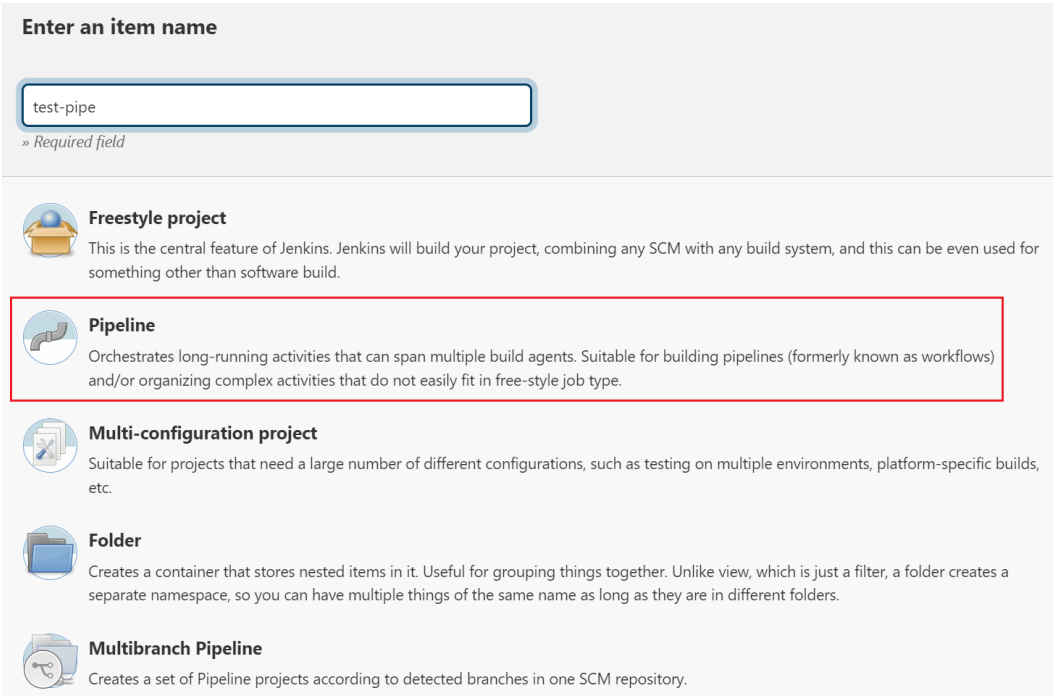
Creating a Pipeline to Build and Push Images

In this example, Jenkins is used to build a pipeline to pull code from the code repository, package the code into an image, and push the image to SWR.

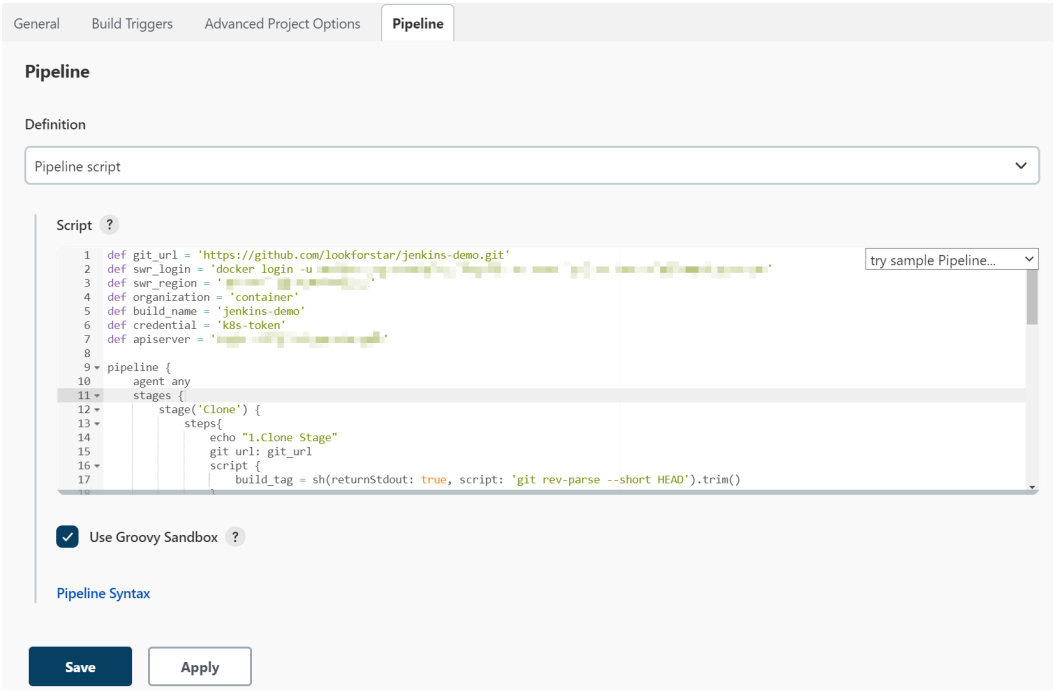
The pipeline creation procedure is as follows:

Step 1 Click **New Item** on the Jenkins page.

Step 2 Enter a task name and select **Create Pipeline**.



Step 3 Configure only the pipeline script.



The following pipeline scripts are for reference only. You can customize the script. For details about the syntax, see **Pipeline**.

Some parameters in the example need to be modified:

- **git_url**: Address of your code repository. Replace it with the actual address.
- **swr_login**: The login command obtained in **Obtaining a Long-Term Valid Login Command**.

- **swr_region**: SWR region.
- **organization**: The actual organization name in SWR.
- **build_name**: Name of the created image.
- **credential**: The cluster credential added to Jenkins. Enter the credential ID. If you want to deploy the service in another cluster, add the access credential of the cluster to Jenkins again. For details, see **Setting Cluster Access Credentials**.
- **apiserver**: IP address of the API server where the application cluster is deployed. Ensure that the IP address can be accessed from the Jenkins cluster.

```
//Define the code repository address.
def git_url = 'https://github.com/lookforstar/jenkins-demo.git'
//Define the SWR login command.

//Define the SWR region.
def swr_region = 'ap-southeast-1'
//Define the name of the SWR organization to be uploaded.
def organization = 'container'
//Define the image name.
def build_name = 'jenkins-demo'
//Certificate ID of the cluster to be deployed
def credential = 'k8s-token'
//API server address of the cluster. Ensure that the address can be accessed from the Jenkins cluster.
def apiserver = 'https://192.168.0.100:6443'

pipeline {
    agent any
    stages {
        stage('Clone') {
            steps{
                echo "1.Clone Stage"
                git url: git_url
                script {
                    build_tag = sh(returnStdout: true, script: 'git rev-parse --short HEAD').trim()
                }
            }
        }
        stage('Test') {
            steps{
                echo "2.Test Stage"
            }
        }
        stage('Build') {
            steps{
                echo "3.Build Docker Image Stage"
                sh "docker build -t swr.${swr_region}.myhuaweicloud.com/${organization}/${build_name}:${build_tag} ."
                //${build_tag} indicates that the build_tag variable is obtained as the image tag. It is the return value of the git rev-parse --short HEAD command, that is, commit ID.
            }
        }
        stage('Push') {
            steps{
                echo "4.Push Docker Image Stage"
                sh swr_login
                sh "docker push swr.${swr_region}.myhuaweicloud.com/${organization}/${build_name}:${build_tag}"
            }
        }
        stage('Deploy') {
            steps{
                echo "5. Deploy Stage"
                echo "This is a deploy step to test"
                script {
                sh "cat k8s.yaml"
                echo "begin to config kubenetes"
```



```
            try {
                withKubeConfig([credentialsId: credential, serverUrl: apiserver]) {
                    sh 'kubectl apply -f k8s.yaml'
                    //The YAML file is stored in the code repository. The following is only an example. Replace
it as required.
                }
                println "hooray, success"
            } catch (e) {
                println "oh no! Deployment failed! "
                println e
            }
            }
        }
    }
  }
}
```

Step 4 Save the settings and execute the Jenkins job.

----End

4.1.3.4 Interconnecting Jenkins with RBAC of Kubernetes Clusters (Example)

Prerequisites

RBAC must be enabled for the cluster.

Scenario 1: Namespace-based Permissions Control

Create a service account and a role, and add a RoleBinding.

```
$ kubectl create ns dev
$ kubectl -n dev create sa dev

$ cat <<EOF > dev-user-role.yml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: dev
  name: dev-user-pod
rules:
- apiGroups: ["*"]
  resources: ["deployments", "pods", "pods/log"]
  verbs: ["get", "watch", "list", "update", "create", "delete"]
EOF
kubectl create -f dev-user-role.yml

$ kubectl create rolebinding dev-view-pod \
    --role=dev-user-pod \
    --serviceaccount=dev:dev \
    --namespace=dev
```

Generate the kubeconfig file of a specified service account.

NOTE

- In clusters earlier than v1.21, a token is obtained by mounting the secret of the service account to a pod. Tokens obtained this way are permanent. This approach is no longer recommended starting from version 1.21. Service accounts will stop auto creating secrets in clusters from version 1.25.

  In clusters of version 1.21 or later, you can use the **TokenRequest** API to **obtain the token** and use the projected volume to mount the token to the pod. Such tokens are valid for a fixed period. When the mounting pod is deleted, the token automatically becomes invalid. For details, see **Service Account Token Security Improvement**.
- If you need a token that never expires, you can also **manually manage secrets for service accounts**. Although a permanent service account token can be manually created, you are advised to use a short-lived token by calling the **TokenRequest** API for higher security.

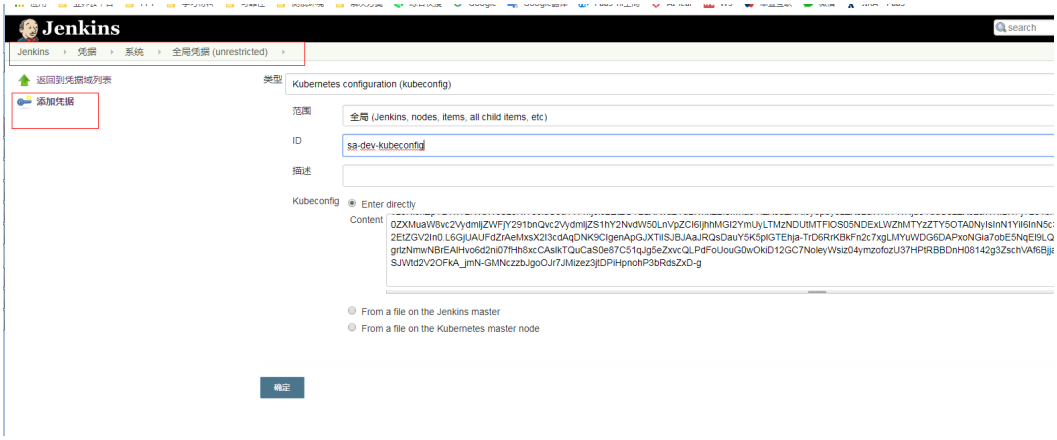
```
$ SECRET=$(kubectl -n dev get sa dev -o go-template='{{range .secrets}}{{.name}}{{end}}')
$ API_SERVER="https://172.22.132.51:6443"
$ CA_CERT=$(kubectl -n dev get secret ${SECRET} -o yaml | awk '/ca.crt:/{print $2}')
$ cat <<EOF > dev.conf
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_CERT
    server: $API_SERVER
  name: cluster
EOF

$ TOKEN=$(kubectl -n dev get secret ${SECRET} -o go-template='{{.data.token}}')
$ kubectl config set-credentials dev-user \
    --token=`echo ${TOKEN} | base64 -d` \
    --kubeconfig=dev.conf

$ kubectl config set-context default \
    --cluster=cluster \
    --user=dev-user \
    --kubeconfig=dev.conf

$ kubectl config use-context default \
    --kubeconfig=dev.conf
```

Verification in the CLI

```
$ kubectl --kubeconfig=dev.conf get po
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:dev:dev" cannot list pods in the namespace "default"

$ kubectl -n dev --kubeconfig=dev.conf run nginx --image nginx --port 80 --restart=Never
$ kubectl -n dev --kubeconfig=dev.conf get po
NAME      READY     STATUS    RESTARTS   AGE
nginx     1/1       Running   0          39s
```

Verify whether the permissions meet the expectation in Jenkins.

Step 1 Add the kubeconfig file with permissions control settings to Jenkins.



Step 2 Start the Jenkins job. In this example, Jenkins fails to be deployed in namespace **default** but is successfully deployed in namespace **dev**.

```
+ cat k8s.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jenkins-demo
  namespace: default
spec:
  template:
    metadata:
      labels:
        app: jenkins-demo
    spec:
      containers:
      - image: cnych/jenkins-demo:716a813
        imagePullPolicy: IfNotPresent
        name: jenkins-demo
        env:
        - name: branch
          value: <BRANCH_NAME>
[Pipeline] echo
begin to config kubenetes
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/liuyi-swr-cce-pipe01/k8s.yaml
ERROR: ERROR: io.fabric8.kubernetes.client.KubernetesClientException: Failure executing: GET at: https://192.168.0.153:5443/apis/extensions/v1beta1/namespaces/default/deployments/jenkins-demo. Message: Forbidden!
User dev-user doesn't have permission. deployments.extensions "jenkins-demo" is forbidden: User "system:serviceaccount:dev:sa-dev" cannot get deployments.extensions in the namespace "default".
hudson.remoting.ProxyException: io.fabric8.kubernetes.client.KubernetesClientException: Failure executing: GET at: https://192.168.0.153:5443/apis/extensions/v1beta1/namespaces/default/deployments/jenkins-demo.
Message: Forbidden! User dev-user doesn't have permission. deployments.extensions "jenkins-demo" is forbidden: User "system:serviceaccount:dev:sa-dev" cannot get deployments.extensions in the namespace "default".
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.requestFailure(OperationSupport.java:472)
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.assertResponseCode(OperationSupport.java:409)
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleResponse(OperationSupport.java:381)
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleResponse(OperationSupport.java:344)
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleGet(OperationSupport.java:313)
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleGet(OperationSupport.java:296)
        at io.fabric8.kubernetes.client.dsl.base.BaseOperation.handleGet(BaseOperation.java:770)
        at io.fabric8.kubernetes.client.dsl.base.BaseOperation.getMandatory(BaseOperation.java:195)
        at io.fabric8.kubernetes.client.dsl.base.BaseOperation.get(BaseOperation.java:162)
```

```
+ cat k8s.yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jenkins-demo
  namespace: dev
spec:
  template:
    metadata:
      labels:
        app: jenkins-demo
    spec:
      containers:
      - image: swr.ap-southeast-2.myhuaweicloud.com/paas-poc/jenkins-demo:f31618e
        imagePullPolicy: IfNotPresent
        name: jenkins-demo
        env:
        - name: branch
          value: <BRANCH_NAME>
[Pipeline] echo
begin to config kubenetes
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/liuyi-swr-cce-pipe01/k8s.yaml
Created Deployment: Deployment(apiVersion=extensions/v1beta1, kind=Deployment, metadata=ObjectMeta(annotations=null, clusterName=null, creationTimestamp=2019-02-18T08:05:47Z, deletionGracePeriodSeconds=null,
deletionTimestamp=null, finalizers=[], generateName=null, generation=1, initializers=null, labels={app=jenkins-demo}, name=jenkins-demo, namespace=dev, ownerReferences=[], resourceVersion=582129,
selfLink=/apis/extensions/v1beta1/namespaces/dev/deployments/jenkins-demo, uid=ffdd740a-3353-11e9-9411-fa163e699047, additionalProperties={}), spec=DeploymentSpec(minReadySeconds=null, paused=null,
progressDeadlineSeconds=null, replicas=1, revisionHistoryLimit=null, rollbackTo=null, selector=LabelSelector(matchExpressions=[], matchLabels={app=jenkins-demo}, additionalProperties={}),
strategy=DeploymentStrategy(rollingUpdate=RollingUpdateDeployment(maxSurge=IntOrString(IntVal=1, Kind=null, StrVal=null, additionalProperties={}), maxUnavailable=IntOrString(IntVal=1, Kind=null, StrVal=null,
additionalProperties={}), additionalProperties={}), type=RollingUpdate, additionalProperties={}), template=PodTemplateSpec(metadata=ObjectMeta(annotations=null, clusterName=null, creationTimestamp=null,
deletionGracePeriodSeconds=null, deletionTimestamp=null, finalizers=[], generateName=null, generation=null, initializers=null, labels={app=jenkins-demo}, name=null, namespace=null, ownerReferences=[],
resourceVersion=null, selfLink=null, uid=null, additionalProperties={}), spec=PodSpec(activeDeadlineSeconds=null, affinity=null, automountServiceAccountToken=null, containers=[Container(args=[], command=[], env=
[EnvVar(name=branch, value=<BRANCH_NAME>, valueFrom=null, additionalProperties={})], envFrom=[], image=swr.ap-southeast-2.myhuaweicloud.com/paas-poc/jenkins-demo:f31618e, imagePullPolicy=IfNotPresent, lifecycle=null,
livenessProbe=null, name=jenkins-demo, ports=[], readinessProbe=null, resources=ResourceRequirements(limits=null, requests=null, additionalProperties={}), securityContext=null, stdin=null, stdinOnce=null,
terminationMessagePath=/dev/termination-log, terminationMessagePolicy=File, tty=null, volumeMounts=[], workingDir=null, additionalProperties={})], dnsPolicy=ClusterFirst, hostAliases=[], hostIPC=null,
hostNetwork=null, hostPID=null, hostname=null, imagePullSecrets=[], initContainers=[], nodeName=null, nodeSelector=null, restartPolicy=Always, schedulerName=default-scheduler,
securityContext=PodSecurityContext(fsGroup=null, runAsNonRoot=null, runAsUser=null, seLinuxOptions=null, supplementalGroups=[], additionalProperties={}), serviceAccount=null, serviceAccountName=null, subdomain=null,
terminationGracePeriodSeconds=30, tolerations=[], volumes=[], additionalProperties={}), additionalProperties={}), status=DeploymentStatus(availableReplicas=null, collisionCount=null,
conditions=[], observedGeneration=null, readyReplicas=null, replicas=null, unavailableReplicas=null, updatedReplicas=null, additionalProperties={}), additionalProperties={})
Finished Kubernetes deployment
[Pipeline] echo
hooray, success
```

----End

Scenario 2: Resource-based Permissions Control

Step 1 Generate the service account, role, and binding.

```
kubectl -n dev create sa sa-test0304
```



```
 cat <<EOF > test0304-role.yml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: dev
  name: role-test0304
rules:
- apiGroups: ["*"]
  resources: ["deployments"]
  resourceNames: ["tomcat03", "tomcat04"]
  verbs: ["get", "update", "patch"]
EOF
kubectl create -f test0304-role.yml

 kubectl create rolebinding test0304-bind \
    --role=role-test0304 \
    --serviceaccount=dev:sa-test0304\
    --namespace=dev
```

Step 2 Generate the kubeconfig file.

NOTE

- In clusters earlier than v1.21, a token is obtained by mounting the secret of the service account to a pod. Tokens obtained this way are permanent. This approach is no longer recommended starting from version 1.21. Service accounts will stop auto creating secrets in clusters from version 1.25.

  In clusters of version 1.21 or later, you can use the **TokenRequest** API to **obtain the token** and use the projected volume to mount the token to the pod. Such tokens are valid for a fixed period. When the mounting pod is deleted, the token automatically becomes invalid. For details, see **Service Account Token Security Improvement**.

- If you need a token that never expires, you can also **manually manage secrets for service accounts**. Although a permanent service account token can be manually created, you are advised to use a short-lived token by calling the **TokenRequest** API for higher security.

```
 SECRET=$(kubectl -n dev get sa sa-test0304 -o go-template='{{range .secrets}}{{.name}}{{end}}')
 API_SERVER=" https://192.168.0.153:5443"
 CA_CERT=$(kubectl -n dev get secret ${SECRET} -o yaml | awk '/ca.crt:/{print $2}')
 cat <<EOF > test0304.conf
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_CERT
    server: $API_SERVER
  name: cluster
EOF

TOKEN=$(kubectl -n dev get secret ${SECRET} -o go-template='{{.data.token}}')
 kubectl config set-credentials test0304-user \
    --token=`echo ${TOKEN} | base64 -d` \
    --kubeconfig=test0304.conf

 kubectl config set-context default \
    --cluster=cluster \
    --user=test0304-user \
    --kubeconfig=test0304.conf

 kubectl config use-context default \
    --kubeconfig=test0304.conf
```

Step 3 Verify that Jenkins is running as expected.

In the pipeline script, update the Deployments of tomcat03, tomcat04, and tomcat05 in sequence.


```
try {
  kubernetesDeploy(
      kubeconfigId: "test0304",
      configs: "test03.yaml")
 println "hooray, success"
} catch (e) {
    println "oh no! Deployment failed! "
    println e
}
echo "test04"
try {
 kubernetesDeploy(
     kubeconfigId: "test0304",
     configs: "test04.yaml")
 println "hooray, success"
} catch (e) {
    println "oh no! Deployment failed! "
    println e
}
echo "test05"
try {
 kubernetesDeploy(
     kubeconfigId: "test0304",
     configs: "test05.yaml")
 println "hooray, success"
} catch (e) {
    println "oh no! Deployment failed! "
    println e
}
```

Viewing the running result:

Figure 4-15 test03

Figure 4-16 test04

```
test04
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/liuyi-swr-cce-pipe01/test04.yaml
Applied Deployment: Deployment(apiVersion=extensions/v1beta1, kind=Deployment, metadata=Obje
deletionTimestamp=null, finalizers=[], generateName=null, generation=3, initializers=null, l
selfLink=/apis/extensions/v1beta1/namespaces/dev/deployments/tomcat04, uid=06af3b14-3356-11e
progressDeadlineSeconds=null, replicas=1, revisionHistoryLimit=null, rollbackTo=null, select
strategy=DeploymentStrategy(rollingUpdate=RollingUpdateDeployment(maxSurge=IntOrString(IntVa
additionalProperties={}), additionalProperties={}), type=RollingUpdate, additionalProperties
deletionGracePeriodSeconds=null, deletionTimestamp=null, finalizers=[], generateName=null, g
resourceVersion=null, selfLink=null, uid=null, additionalProperties={}), spec=PodSpec(active
[EnvVar(name=branch, value=<BRANCH_NAME>, valueFrom=null, additionalProperties={})], envFrom
livenessProbe=null, name=tomcat04, ports=[], readinessProbe=null, resources=ResourceRequirem
terminationMessagePath=/dev/termination-log, terminationMessagePolicy=File, tty=null, volume
hostNetwork=null, hostPID=null, hostname=null, imagePullSecrets=[], initContainers=[], nodeN
securityContext=PodSecurityContext(fsGroup=null, runAsNonRoot=null, runAsUser=null, seLinuxO
terminationGracePeriodSeconds=30, tolerations=[], volumes=[], additionalProperties={}), addi
conditions=[DeploymentCondition(lastTransitionTime=2019-02-18T08:56:55Z, lastUpdateTime=2019
type=Available, additionalProperties={}), DeploymentCondition(lastTransitionTime=2019-02-18T
reason=ReplicaSetUpdated, status=True, type=Progressing, additionalProperties={})], observed
additionalProperties={})
Finished Kubernetes deployment
[Pipeline] echo
hooray, success
[Pipeline] echo
test05
[Pipeline] kubernetesDeploy
Starting Kubernetes deployment
Loading configuration: /var/jenkins_home/workspace/liuyi-swr-cce-pipe01/test05.yaml
ERROR: ERROR: io.fabric8.kubernetes.client.KubernetesClientException: Failure executing: GET
test0304-user doesn't have permission. deployments.extensions "tomcat05" is forbidden: User
hudson.remoting.ProxyException: io.fabric8.kubernetes.client.KubernetesClientException: Fail
Forbidden! User test0304-user doesn't have permission. deployments.extensions "tomcat05" is
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.requestFailure(OperationSu
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.assertResponseCode(Operati
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleResponse(OperationSu
        at io.fabric8.kubernetes.client.dsl.base.OperationSupport.handleResponse(OperationSu
```

----End

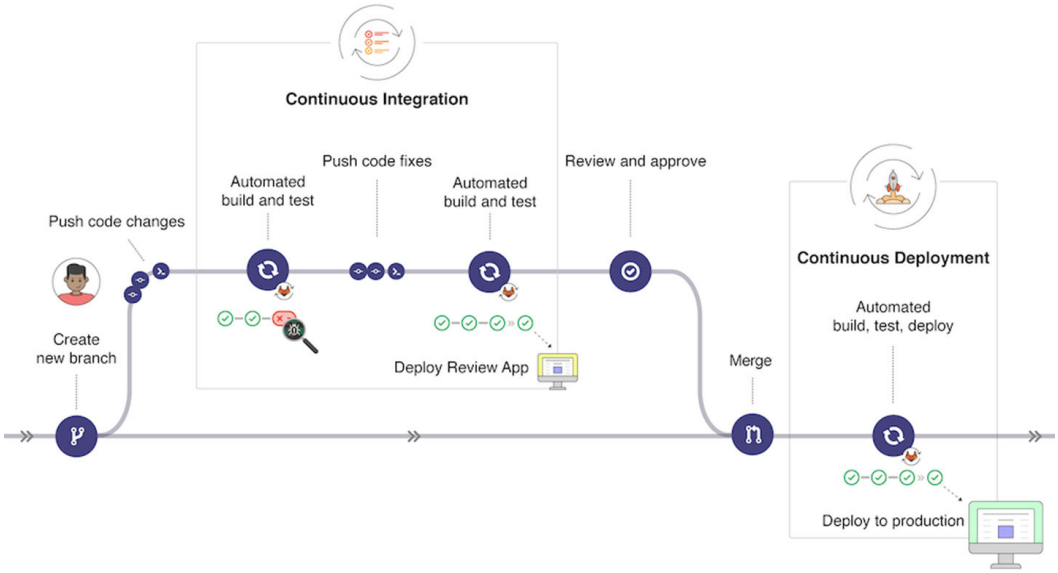
4.2 Interconnecting GitLab with SWR and CCE for CI/CD

Background

GitLab is an open-source version management system developed with Ruby on Rails for Git project repository management. It supports web-based access to public and private projects. Similar to GitHub, GitLab allows you to browse source code, manage bugs and comments, and control team member access to repositories. You will find it very easy to view committed versions and file history database. Team members can communicate with each other using the built-in chat program (Wall).

GitLab provides powerful CI/CD functions and is widely used in software development.

Figure 4-17 GitLab CI/CD process

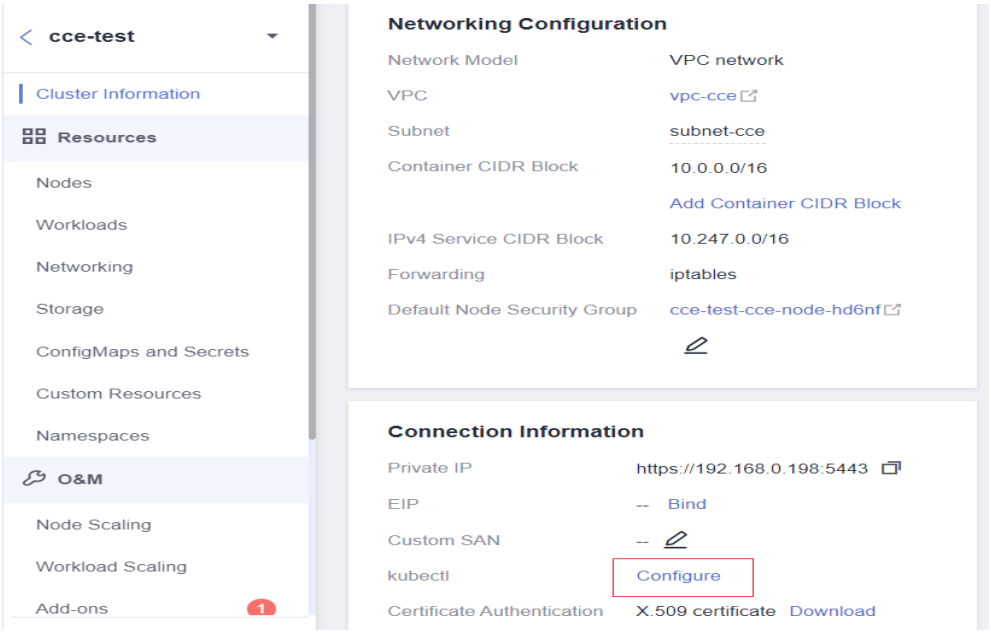


This section describes how to interconnect GitLab with SWR and CCE for CI/CD.

Preparations

1. Create a CCE cluster and a node and bind an EIP to the node for downloading an image during GitLab Runner installation.
2. Download and configure kubectl to connect to the cluster.

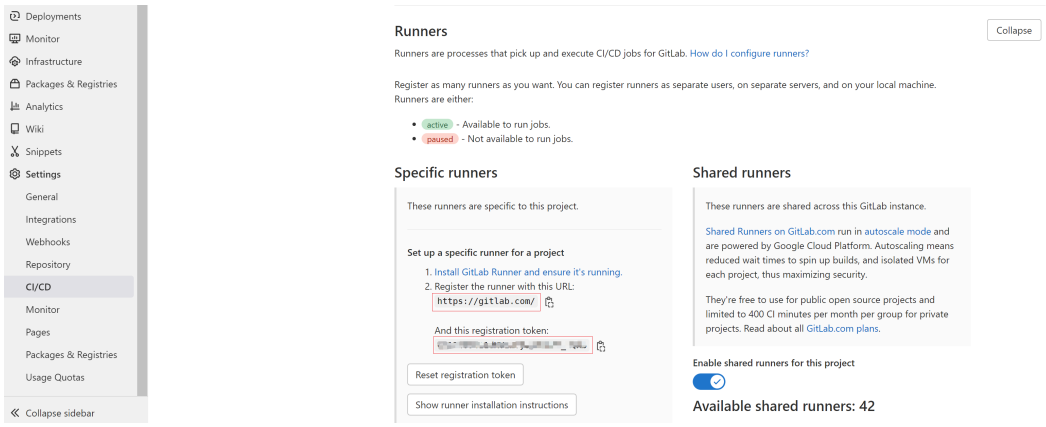
   Log in to the CCE console, click **Configure** for kubectl in the **Connection Information** pane on the cluster console, and configure kubectl as instructed.



3. **Install Helm 3**.

Installing GitLab Runner

Log in to **GitLab**, choose **Settings** > **CI/CD** in the project view, click **Expand** next to **Runners**, and search for the GitLab Runner registration URL and token.



Create the **values.yaml** file and fill in the following information:

```
# Registration URL
gitlabUrl: https://gitlab.com/
# Registration token
runnerRegistrationToken: "******"
rbac:
    create: true
runners:
    privileged: true
```

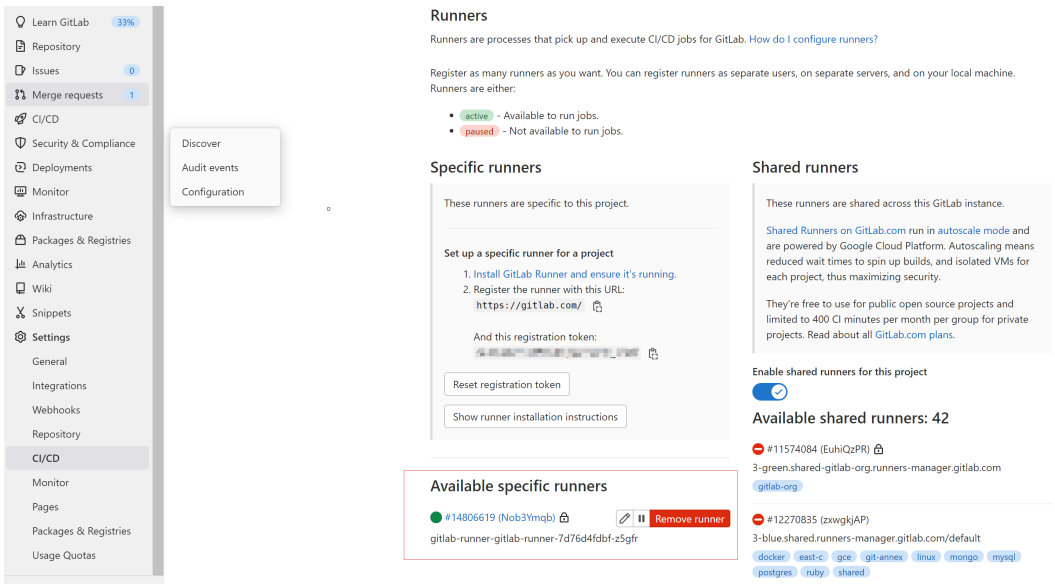
Create a GitLab namespace.

```
kubectl create namespace gitlab
```

Install GitLab Runner using Helm.

```
helm repo add gitlab https://charts.gitlab.io
helm install --namespace gitlab gitlab-runner -f values.yaml gitlab/gitlab-runner --version=0.43.1
```

After the installation, you can obtain the gitlab-runner workload on the CCE console and view the connection information in GitLab later.



Creating an Application

Place the application to be created in the GitLab project repository. This section takes Nginx modification as an example. For details, visit **https://gitlab.com/c8147/cidemo/-/tree/main**.

The following files are included:

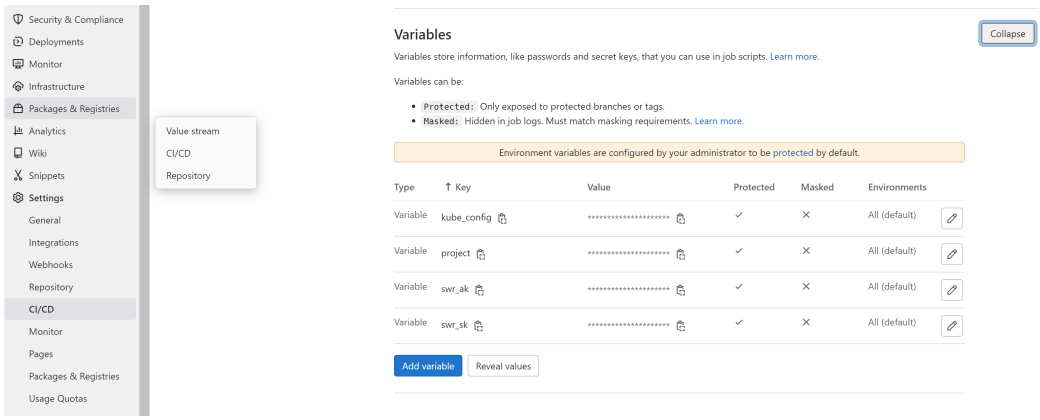
- **.gitlab-ci.yml**: Gitlab CI file, which will be described in detail in **Creating a Pipeline**.
- **Dockerfile**: used to build Docker images.
- **index.html**: used to replace the index page of Nginx.
- **k8s.yaml**: used to deploy the Nginx app. A Deployment named **nginx-test** and a Service named **nginx-test** will be created.

The preceding files are only examples. You can replace or modify them accordingly.

Configuring Global Variables

When using pipelines, build an image, upload it to SWR, and run kubectl commands to deploy the image in the cluster. Before performing these operations, you must log in to SWR and obtain the credential for connecting to the cluster. You can define the information as variables in GitLab.

Log in to **GitLab**, choose **Settings** > **CI/CD** in the project view, and click **Expand** next to **Variables** to add variables.



- **kube_config**

  kubeconfig.json file used for kubectl command authentication. Run the following command on the host where kubectl is configured to convert the file to the Base64 format:

  echo $(cat ~/.kube/config | base64) | tr -d " "

  The command output is the content of **kubeconfig.json**.

- **project**: project name.

  Log in to the management console, click your username in the upper right corner, and click **My Credentials**. In the **Projects** area on the **API Credentials** page, check the name of the project in your current region.

- **swr_ak**: access key.

Log in to the management console, click your username in the upper right corner, and click **My Credentials**. In the navigation pane, choose **Access Keys**. Click **Create Access Key**, enter the description, and click **OK**. In the displayed **Information** dialog box, click **Download**. After the certificate is downloaded, obtain the AK and SK information from the **credentials** file.

- **swr_sk**: secret key for logging in to SWR.

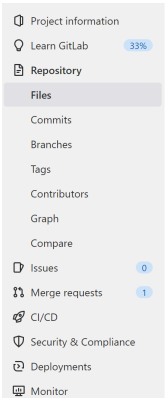
  Run the following command to obtain the key pair. Replace *$AK* and *$SK* with the AK and SK obtained in the preceding steps.

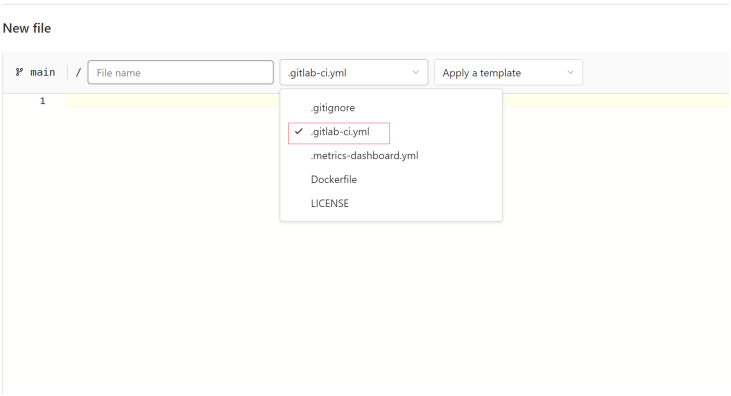
  printf "*$AK*" | openssl dgst -binary -sha256 -hmac "*$SK*" | od -An -vtx1 | sed 's/[\n]//g' | sed 'N;s/\n//'

  The command output displays the login key pair.

Creating a Pipeline

Log in to **Gitlab** and add the **.gitlab-ci.yml** file to **Repository**.





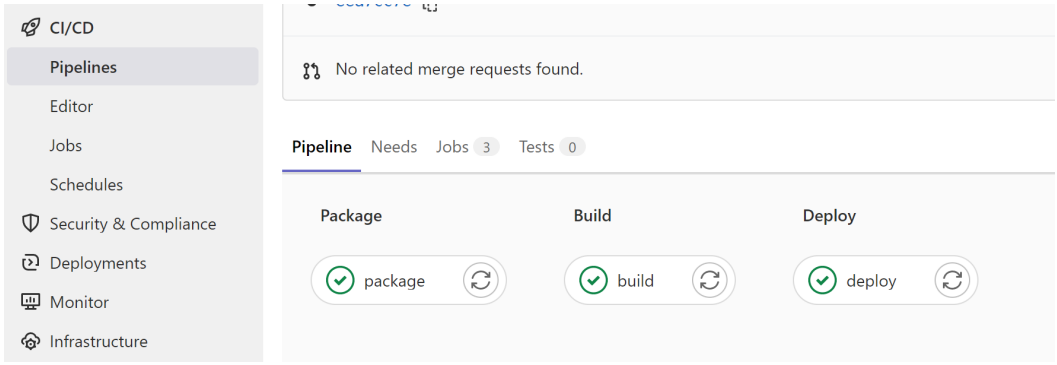
The content is as follows:

```
# Define pipeline stages, including package, build, and deploy.
stages:
  - package
  - build
  - deploy
# If no image is specified in each stage, the default image docker:latest is used.
image: docker:latest
# In the package stage, only printing is performed.
package:
  stage: package
  script:
    - echo "package"
# In the build stage, the Docker-in-Docker mode is used.
build:
  stage: build
  # Define environment variables for the build stage.
  variables:
    DOCKER_HOST: tcp://docker:2375
  # Define the image for running Docker-in-Docker.
  services:
    - docker:18.09-dind
  script:
    - echo "build"
    # Log in to SWR.
    - docker login -u $project@$swr_ak -p $swr_sk swr.ap-southeast-1.myhuaweicloud.com
    # Build an image. k8s-dev is the organization name in SWR. Replace it to the actual name.
    - docker build -t swr.ap-southeast-1.myhuaweicloud.com/k8s-dev/nginx:$CI_PIPELINE_ID .
    # Push the image to SWR.
    - docker push swr.ap-southeast-1.myhuaweicloud.com/k8s-dev/nginx:$CI_PIPELINE_ID
```



```
deploy:
  # Use the kubectl image.
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  stage: deploy
  script:
    # Configure the kubeconfig file.
    - mkdir -p $HOME/.kube
    - export KUBECONFIG=$HOME/.kube/config
    - echo $kube_config |base64 -d > $KUBECONFIG
    # Replace the image in the k8s.yaml file.
    - sed -i "s/<IMAGE_NAME>/swr.ap-southeast-1.myhuaweicloud.com\/k8s-dev\/nginx:$CI_PIPELINE_ID/g" k8s.yaml
    - cat k8s.yaml
    # Deploy an application.
    - kubectl apply -f k8s.yaml
```

After the **.gitlab-ci.yml** file is saved, the pipeline is started immediately. You can view the pipeline execution status in GitLab.



Verifying Deployment

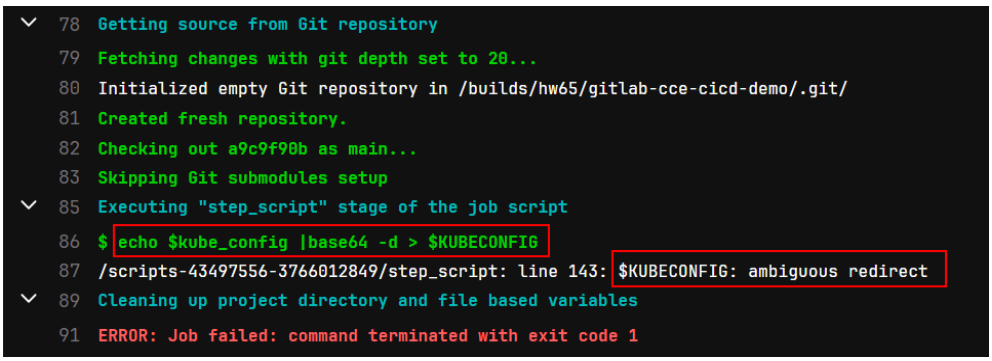
After the pipeline is deployed, locate the **nginx-test** Service on the CCE console, query its access address, and run the **curl** command to access the Service.

```
# curl xxx.xxx.xxx.xxx:31111
Hello Gitlab!
```

If the preceding information is displayed, the deployment is correct.

Common Issues

- If the following problem occurs during the deployment:



  Or


```
$ kubectl apply -f k8s.yaml
E0215 08:03:55.297105      19 memcache.go:255] couldn't get resource list for proxy.exporter.k8s.io/v1beta1: Got empty response for: proxy.exporter.k8s.io/v1beta1
Error from server (Forbidden): error when retrieving current configuration of:
Resource: "apps/v1, Resource=deployments", GroupVersionKind: "apps/v1, Kind=Deployment"
Name: "nginx-test", Namespace: "gitlab"
from server for: "k8s.yaml": deployments.apps "nginx-test" is forbidden: User "system:serviceaccount:gitlab:default" cannot get resource "deployments" in API group "apps" in the namespace "gitlab"
Error from server (Forbidden): error when retrieving current configuration of:
Resource: "/v1, Resource=services", GroupVersionKind: "/v1, Kind=Service"
Name: "nginx-test", Namespace: "gitlab"
from server for: "k8s.yaml": services "nginx-test" is forbidden: User "system:serviceaccount:gitlab:default" cannot get resource "services" in API group "" in the namespace "gitlab"
Cleaning up project directory and file based variables
00:01
ERROR: Job failed: command terminated with exit code 1
```

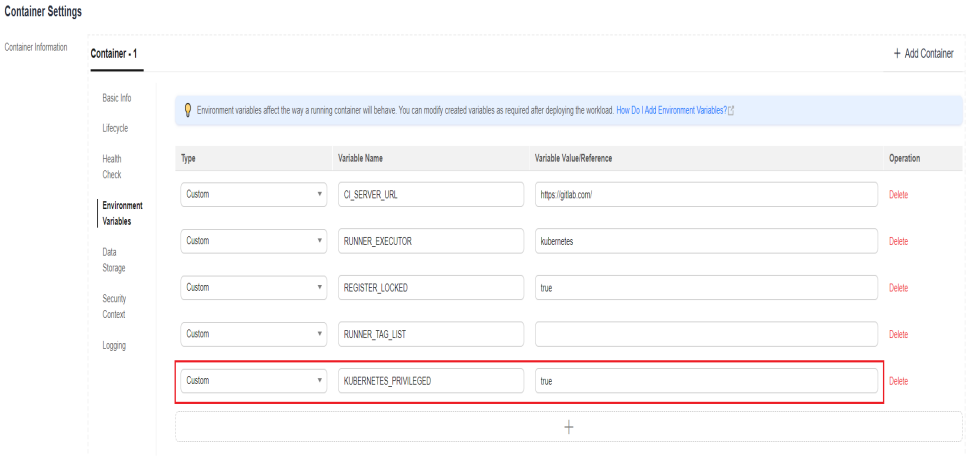
Check whether the following commands are missing in the **.gitlab-ci.yml** file. If yes, add them to the **.gitlab-ci.yml** file.

```
...
deploy:
  # Use the kubectl image.
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  stage: deploy
  script:
    # Configure the kubeconfig file.
    - mkdir -p $HOME/.kube
    - export KUBECONFIG=$HOME/.kube/config
    - echo $kube_config |base64 -d > $KUBECONFIG
    # Replace the image in the k8s.yaml file.
...
```

- If Docker cannot be executed, information similar to the following will display.

```
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
Login Succeeded
$ docker build -t swr.ap-southeast-2.myhuaweicloud.com/wpf-test/nginx:$CI_PIPELINE_ID .
Cannot connect to the Docker daemon at tcp://docker:2375. Is the docker daemon running?
Cleaning up project directory and file based variables
00:00
ERROR: Job failed: command terminated with exit code 1
```

The **privileged: true** parameter fails to be transferred during GitLab Runner installation. As a result, you do not have the permission to run the docker command. To resolve this issue, find GitLab Runner in the workload list on the CCE console, add the environment variable **KUBERNETES_PRIVILEGED**, and set its value to **true**.



5 Disaster Recovery

5.1 Recommended Configurations for Cluster HA

This section describes the recommended configurations for a Kubernetes cluster in which applications can run stably and reliably.

Item	Description	Recommended Operations
Master node	CCE is a hosted Kubernetes cluster service. You do not need to perform O&M on the master nodes. You can configure your cluster specifications to improve the stability and reliability.	• **Deploying the Master Nodes in Different AZs** • **Selecting a Network Model** • **Selecting a Service Forwarding Mode** • **Configuring Quotas and Limits for the Cloud Service Resources and Resources in a Cluster** • **Monitoring Metrics of the Master Nodes**
Worker node	In a Kubernetes cluster, the data plane consists of worker nodes that can run containerized applications and transmit network traffic. When using CCE, perform O&M on worker nodes by yourself. To achieve HA, ensure the worker nodes' scalability and repairability and pay attention to the running statuses of the worker nodes' key components.	• **Partitioning Data Disks Attached to a Node** • **Running npd** • **Configuring the DNS Cache** • **Properly Deploying CoreDNS**

Item	Description	Recommended Operations
Application	If you want your applications to be always available, especially during peak hours, run them in a scalable and elastic manner and pay attention to their running statuses.	• **Running Multiple Pods** • **Configuring Resource Quotas for a Workload** • **Deploying an Application in Multiple AZs** • **Deploying an Add-on in Multiple AZs** • **Configuring Auto Scaling** • **Viewing Logs, Monitoring Metrics, and Adding Alarm Rules**

Deploying the Master Nodes in Different AZs

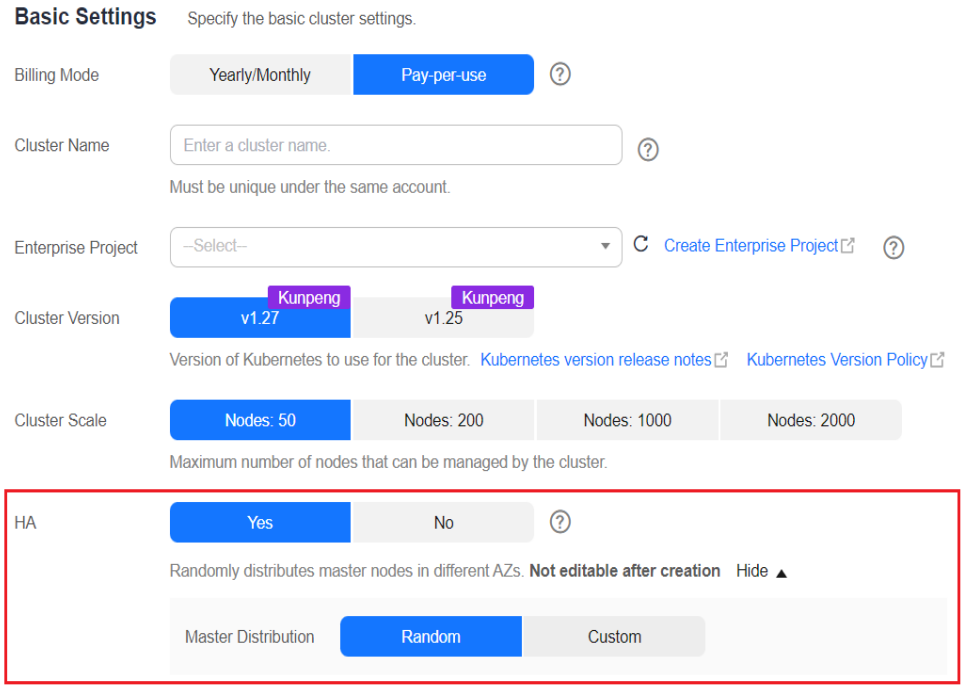
Multiple regions are provided for you to deploy your services, and there are different availability zones (AZs) in each region. An AZ is a collection of one or more physical data centers with independent cooling, fire extinguishing, moisture-proof, and electricity facilities in each AZ. AZs within a region are connected using high-speed optical fibers. This allows you to build cross-AZ HA systems.

When creating a cluster, enable the HA mode of the cluster and configure the distribution mode of the master nodes. The master nodes are randomly deployed in different AZs. This ensures a higher disaster recovery (DR) capability of the cluster.

You can also customize the distribution mode. The following two modes are supported:

- **Random**: Master nodes are deployed in different AZs for DR.
- **Custom**: Master nodes are deployed in specific AZs.
  - **Host**: Master nodes are deployed on different hosts in the same AZ.
  - **Custom**: Master nodes are deployed in the AZ you specify.

Figure 5-1 Configuring the distribution mode of the master nodes



Selecting a Network Model

- Network model: CCE supports VPC network, Cloud Native 2.0 network, and container tunnel network models for your clusters. Different models have different performance and functions. For details, see **Network Models**.
- VPC network: To enable your applications to access other cloud services like RDS, create related services in the same VPC network as your cluster which runs these applications. This is because services using different VPC networks are isolated from each other. If you have created instances, use **VPC peering** to enable communication between VPCs.
- Container CIDR block: Do not configure a small container CIDR block. Otherwise, the number of supported nodes will be limited.
  - For a cluster using a VPC network, if the subnet mask of the container CIDR block is /16, there are 256 x 256 IP addresses available. If the maximum number of pods reserved on each node is 128, the maximum number of nodes supported is 512.
  - For a cluster using a container tunnel network, if the subnet mask of the container CIDR block is /16, there are 256 x 256 IP addresses assigned to your cluster. The container CIDR block allocates 16 IP addresses to the nodes at a time by default. The maximum number of nodes supported by your cluster is 4096 (65536/16=4096).
  - For a cluster using the Cloud Native Network 2.0, the container CIDR block is the VPC subnet, and the number of containers can be created depends on the size of the selected subnet.
- Service CIDR block: The service CIDR block determines the upper limit of Service resources in your cluster. Evaluate your actual needs and then configure the CIDR block. A created CIDR block cannot be modified. Do not configure an excessively small one.

For details, see **Planning CIDR Blocks for a Cluster**.

Selecting a Service Forwarding Mode

kube-proxy is a key component of a Kubernetes cluster. It is responsible for load balancing and forwarding between a Service and its backend pod. When using clusters, consider the potential performance problems of the forwarding mode.

CCE supports the iptables and IPVS forwarding modes.

- IPVS allows higher throughput and faster forwarding. It applies to scenarios where the cluster scale is large or the number of Services is large.
- iptables is the traditional kube-proxy mode. This mode applies to the scenario where the number of Services is small or there are a large number of short concurrent connections on the client. When there are more than 1000 Services in the cluster, network delay may occur.

Configuring Quotas and Limits for the Cloud Service Resources and Resources in a Cluster

CCE allows you to configure resource quotas and limits for your cloud service resources and resources in your clusters. This prevents excessive use of resources. When creating your applications for CCE clusters, consider these limits and periodically review them. This will avoid scaling failures caused by insufficient quotas during application running.

- Configuring resource quotas for cloud services: Cloud services like ECS, EVS, VPC, ELB, and SWR are also used to run the CCE clusters. If the existing resource quotas cannot meet your requirements, submit a service ticket to increase the quotas.
- Configuring resource quotas for a cluster: You are allowed to configure the namespace-level resource quotas to limit the number of objects of a certain type created in a namespace and the total computing resources like CPU and memory consumed by the objects. For details, see **Configuring Resource Quotas**.

Monitoring Metrics of the Master Nodes

Monitoring metrics of the master nodes allows you to check the master nodes' performance and efficiently identify problems occurred on them. The master nodes which are not running properly may lower application reliability.

CCE allows you to monitor kube-apiserver, kube-controller, kube-scheduler, and etcd-server on the master nodes with the **kube-prometheus-stack** add-on installed. With grafana, you can use the **Kubernetes monitoring overview dashboard** to monitor metrics of Kubernetes API server requests and latency and etcd latency.

If your prometheus add-on is used, you can manually add monitoring metrics. For details, see **Monitoring Metrics of the Master Node Components**.

Partitioning Data Disks Attached to a Node

By default, the first data disk of a worker node is for storing the container runtime and kubelet components. The remaining capacity of this data disk affects image

download and container startup and running. For details, see **Data Disk Space Allocation**.

The default space of this date disk is 100 GiB. You can adjust the space as required. Images, system logs, and application logs are stored on data disks. Therefore, you need to evaluate the number of pods to be deployed on each node, the size of logs, images, and temporary data of each pod, as well as some reserved space for the system. For details, see **Selecting a Data Disk for the Node**.

Running npd

A failure in a worker node may affect the availability of the applications. **npd** is used to monitor node exceptions. It helps you detect and handle latent exceptions in a timely manner. You can also customize the check items, including target node, check period, and triggering threshold. For details, see **Node Fault Detection Policy**.

Configuring the DNS Cache

When the number of DNS requests in a cluster increases, the load of CoreDNS increases and the following issues may occur:

- Increased delay: CoreDNS needs to process more requests, which may slow down the DNS query and affect service performance.
- Increased resource usage: To ensure DNS performance, CoreDNS requires higher specifications.

To minimize the impact of DNS delay, deploy NodeLocal DNSCache in the cluster to improve the networking stability and performance. NodeLocal DNSCache runs a DNS cache proxy on cluster nodes. All pods with DNS configurations use the DNS cache proxy running on nodes instead of the CoreDNS service for domain name resolution. This reduces CoreDNS' load and improves the cluster DNS performance.

To deploy NodeLocal DNSCache, install the **NodeLocal DNSCache**. For details, see **Using NodeLocal DNSCache to Improve DNS Performance**.

Properly Deploying CoreDNS

Deploy the CoreDNS instances in different AZs and nodes to mitigate the single-node or single-AZ faults.

Ensure that the CPU and memory of the node where CoreDNS is running are not fully used. Otherwise, the Queries per second (QPS) and response of domain name resolution will be affected.

For details about how to properly configure CoreDNS, see **Configuring CoreDNS**.

Running Multiple Pods

If your application runs in one pod, the application will be unavailable if the pod is abnormal. Use Deployments or other types of replicas to deploy your applications. Each time a pod fails or is terminated, the controller automatically restarts a new pod that has the same specifications as the original one to ensure that a specified number of pods are always running in the cluster.

When creating a workload, set the number of instances to a value greater than 2. If an instance is faulty, the remaining instances still run until Kubernetes automatically creates another pod to compensate for the loss. You can also use HPA and CA (**Using HPA and CA for Auto Scaling of Workloads and Nodes**) to automatically scale in or out the workloads as required.

Using Containers to Isolate Processes

Containers provide process-level isolation. Each container has its own file system, network, and resource allocation. This prevents interference between different processes and avoids attacks and data leakage from malicious processes. Using containers to isolate processes can improve the reliability, security, and portability of applications.

If several processes work together, create multiple containers in a pod so that they can share the same network, PV, and other resources. Taking the init container as an example. The init containers run before the main containers are started to complete some initialization tasks like configuring environment variables, loading databases or data stores, and pulling Git repositories.

Note that multiple containers in a pod share the lifecycle of this pod. Therefore, if one container is abnormal, the entire pod will be restarted.

Configuring Resource Quotas for a Workload

Configure and adjust resource requests and limits for all workloads.

If too many pods are scheduled to one node, the node will be overloaded and unable to provide services.

To avoid this problem, when deploying a pod, specify the request and limit resources required by the pod. Kubernetes then selects a node with sufficient idle resources for this pod. In the following example, the Nginx pod requires 1-core CPU and 1024 MiB memory. The actual usage cannot exceed 2-core CPU and 4096 MiB memory.

Kubernetes statically schedules resources. The remaining resources on each node are calculated as follows: Remaining resources on a node = Total resources on the node – Allocated resources (not resources in use). If you manually run a resource-consuming process, Kubernetes cannot detect it.

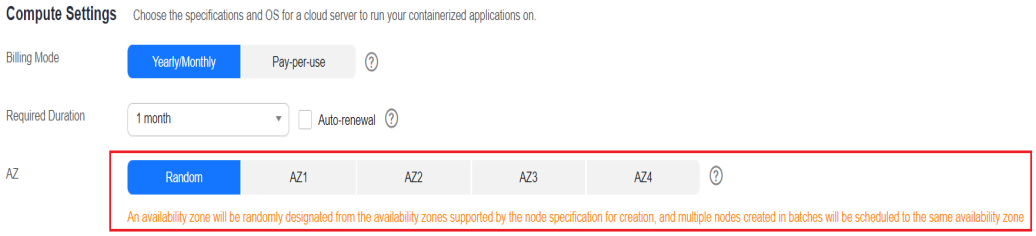
Additionally, the resource usage must be claimed for all pods. For a pod that does not claim the resource usage, after it is scheduled to a node, Kubernetes does not deduct the resources used by this pod from the node on which it is running. Other pods may still be scheduled to this node.

Deploying an Application in Multiple AZs

You can run pods on nodes in multiple AZs to prevent an application from being affected by faults of a single AZ.

When creating a node, manually specify an AZ for the node.

Figure 5-2 Specifying an AZ



During application deployment, configure anti-affinity policies for pods so that the scheduler can schedule pods across multiple AZs. For details, see **Implementing High Availability for Applications in CCE**. The following is an example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      containers:
      - name: web-app
        image: nginx
      imagePullSecrets:
      - name: default-secret
      affinity:
        podAntiAffinity:  # Workload anti-affinity
          preferredDuringSchedulingIgnoredDuringExecution:  # Indicates that the rule is met as much as possible. Otherwise, scheduling cannot be performed when the number of pods exceeds the number of AZs.
          - podAffinityTerm:
              labelSelector:  # Pod label matching rule. Configure anti-affinity policies between pods and their own labels.
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: topology.kubernetes.io/zone  # Topology domain of the AZ where the node is located
            weight: 100
```

You can also use **Pod Topology Spread Constraints** to deploy pods in multiple AZs.

Deploying an Add-on in Multiple AZs

The Deployment pods of CCE system add-ons like CoreDNS and Everest can be deployed in multiple AZs, the same way as deploying an application. This function can satisfy different user requirements.

Table 5-1 Deployment description

Mode	Configuration Description	Usage Description	Recommended Configuration Scenario
Preferred	Add-on pods will have labels with the key **topology.kubernetes.io/zone** for soft anti-affinity deployment, and the anti-affinity type is **preferredDuringSchedulingIgnoredDuringExecution**.	Add-on pods will be preferentially scheduled to nodes in different AZs. If resources in some AZs are insufficient, some add-on pods may be scheduled to the same AZ which has sufficient resources.	No mandatory requirements for multi-AZ DR
Required	Add-on pods will have labels with the key **topology.kubernetes.io/zone** for hard anti-affinity deployment, and the anti-affinity type is **requiredDuringSchedulingIgnoredDuringExecution**.	A maximum of one pod of the same add-on can be deployed in each AZ. The number of running pods cannot exceed the number of AZs in the cluster. If the node where the add-on pod runs is faulty, pods running on the faulty node cannot be automatically migrated to other nodes in the same AZ.	Changing number of AZs (This mode is used to prevent all pods from being scheduled to the node in the current AZ in advance.)

Mode	Configuration Description	Usage Description	Recommended Configuration Scenario
Equivalent mode	Add-on pods will have labels with the key **topology.kubernetes.io/zone** for configuring topology spread constraints. The pod difference between different topology domains cannot exceed 1 for add-on pods to be evenly distributed in different AZs.	The effect of this mode is between that of the preferred mode and that of the required mode. In the equivalent mode, add-on pods can be deployed in different AZs. Additionally, multiple pods can be deployed in a single AZ when there are more pods than AZs. To use this mode, you need to plan node resources in each AZ in advance to ensure that each AZ has sufficient node resources for deploying pods. (If there are more than 1 add-on pods in a single AZ, the nodes to which the add-on pods can be scheduled in each AZ should be one more than the actual add-on pods in the current AZ.) This ensures successful deployment of add-on pods although node resources in some AZ are insufficient and smooth scheduling of add-on pods during update.	Scenarios have high requirements for DR

Configuring Health Check for a Container

Kubernetes automatically restarts pods that are not running properly. This prevents service interruption caused by exceptions of pods. In some cases, however, even if a pod is running, it does not mean that it can provide services properly. For example, a deadlock may occur in a process in a running pod, but Kubernetes does not automatically restart the pod because it is still running. To solve this problem, configure a liveness probe to check whether the pod is healthy. If the liveness probe detects a problem, Kubernetes will restart the pod.

You can also configure a readiness probe to check whether the pod can provide normal services. After an application container is started, it may take some time for initialization. During this process, the pod on which this container is running cannot provide services to external systems. The Services forward requests to this pod only when the readiness probe detects that the pod is ready. When a pod is

faulty, the readiness probe can prevent new traffic from being forwarded to the pod.

The startup probe is used to check whether the application container is started. The startup probe ensures that the containers can start successfully before the liveness probe and readiness probe do their tasks. This ensures that the liveness probe and readiness probe do not affect the startup of containers. Configuring the startup probe ensures that the slow-start containers can be detected by the liveness probe to prevent Kubernetes from terminating them before they are started.

You can configure the preceding probes when creating an application. The following is an example:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: nginx:alpine
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      exec:
        command:
          - cat
          - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
    startupProbe:
      httpGet:
        path: /healthz
        port: 80
      failureThreshold: 30
      periodSeconds: 10
```

For details, see **Configuring Container Health Check**.

Configuring Auto Scaling

Auto scaling can automatically adjust the number of application containers and nodes as required. Containers and nodes can be quickly scaled out or scaled in to save resources and costs.

Typically, two types of auto scaling may occur during peak hours:

- Workload scaling: When pods or containers are used for deploying applications, the requested and limit values of the containers are generally configured to prevent unlimited usage of resources during peak hours. However, after the upper limit is reached, an application error may occur. To resolve this issue, scale in the number of pods to share workloads.

- Node scaling: After the number of pods grows, the resource usage of the node may increase to a certain extent. This results in that the added pods cannot be scheduled. To solve this problem, scale in or out nodes based on the resource usage.

For details, see **Using HPA and CA for Auto Scaling of Workloads and Nodes**.

Viewing Logs, Monitoring Metrics, and Adding Alarm Rules

- Logging
  - Control plane logs are reported from the master nodes. CCE supports kube-controller-manager, kube-apiserver, kube-scheduler, and audit logs. For details, see **Collecting Control Plane Component Logs**.
  - Application logs are generated by pods. These logs include logs generated by pods in which the service containers are running and Kubernetes system components like CoreDNS. CCE allows you to configure policies for collecting, managing, and analyzing logs periodically to prevent logs from being over-sized. For details, see **Logging Overview**.
- Monitoring
  - Metrics of the master nodes: Monitoring these metrics enables you to efficiently identify problems occurred on the master nodes. For details, see **Monitoring Metrics of the Master Nodes**.
  - Metrics of the applications: CCE can comprehensively monitor applications in clusters by checking these metrics. In addition to standard metrics, you can configure custom metrics of your applications that comply with their specifications to improve the observability. For details, see **Collecting Container Logs**.
- Alarm

  You can add alarm rules for metrics to detect cluster faults and generate warnings in a timely manner with the monitoring function. This helps you maintain service stability. For details, see **Customized Alarm Configurations**.

5.2 Implementing High Availability for Applications in CCE

Basic Principles

To achieve high availability for your CCE containers, you can do as follows:

1. Deploy three master nodes for the cluster.
2. Create nodes in different AZs. When nodes are deployed across AZs, you can customize scheduling policies based on your requirements to maximize resource utilization.
3. Create multiple node pools in different AZs and use them for node scaling.
4. Set the number of pods to be greater than 2 when creating a workload.
5. Set pod affinity rules to distribute pods to different AZs and nodes.

Procedure

Assume that there are four nodes in a cluster distributed in different AZs.

```
$ kubectl get node -L topology.kubernetes.io/zone,kubernetes.io/hostname
NAME           STATUS   ROLES    AGE   VERSION                   ZONE     HOSTNAME
192.168.5.112   Ready    <none>   42m   v1.21.7-r0-CCE21.11.1.B007   zone01   192.168.5.112
192.168.5.179   Ready    <none>   42m   v1.21.7-r0-CCE21.11.1.B007   zone01   192.168.5.179
192.168.5.252   Ready    <none>   37m   v1.21.7-r0-CCE21.11.1.B007   zone02   192.168.5.252
192.168.5.8     Ready    <none>   33h   v1.21.7-r0-CCE21.11.1.B007   zone03   192.168.5.8
```

Create workloads according to the following podAntiAffinity rules:

- Pod anti-affinity in an AZ. Configure the parameters as follows:
  - **weight**: A larger weight value indicates a higher priority of scheduling. In this example, set it to **50**.
  - **topologyKey**: includes a default or custom key for the node label that the system uses to denote a topology domain. A topology key determines the scope where the pod should be scheduled to. In this example, set this parameter to **topology.kubernetes.io/zone**, which is the label for identifying the AZ where the node is located.
  - **labelSelector**: Select the label of the workload to realize the anti-affinity between this container and the workload.
- The second one is the pod anti-affinity in the node hostname. Configure the parameters as follows:
  - **weight**: Set it to **50**.
  - **topologyKey**: Set it to **kubernetes.io/hostname**.
  - **labelSelector**: Select the label of the pod, which is anti-affinity with the pod.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 50
              podAffinityTerm:
                labelSelector:                  # Select the label of the workload to realize the anti-affinity
between this container and the workload.
```



```
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - nginx
              namespaces:
                - default
              topologyKey: topology.kubernetes.io/zone   # It takes effect in the same AZ.
            - weight: 50
              podAffinityTerm:
                labelSelector:                    # Select the label of the workload to realize the anti-affinity between this container and the workload.
                  matchExpressions:
                    - key: app
                      operator: In
                      values:
                        - nginx
                namespaces:
                  - default
                topologyKey: kubernetes.io/hostname     # It takes effect on the node.
      imagePullSecrets:
        - name: default-secret
```

Create a workload and view the node where the pod is located.

```
$ kubectl get pod -owide
NAME                  READY   STATUS    RESTARTS   AGE   IP          NODE
nginx-6fffd8d664-dpwbk   1/1     Running   0          17s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-qhclc   1/1     Running   0          17s   10.0.1.133   192.168.5.252
```

Increase the number of pods to 3. The pod is scheduled to another node, and the three nodes are in three different AZs.

```
$ kubectl scale --replicas=3 deploy/nginx
deployment.apps/nginx scaled
$ kubectl get pod -owide
NAME                  READY   STATUS    RESTARTS   AGE     IP          NODE
nginx-6fffd8d664-8t7rv   1/1     Running   0          3s      10.0.0.9    192.168.5.8
nginx-6fffd8d664-dpwbk   1/1     Running   0          2m45s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-qhclc   1/1     Running   0          2m45s   10.0.1.133   192.168.5.252
```

Increase the number of pods to 4. The pod is scheduled to the last node. With podAntiAffinity rules, pods can be evenly distributed to AZs and nodes.

```
$ kubectl scale --replicas=4 deploy/nginx
deployment.apps/nginx scaled
$ kubectl get pod -owide
NAME                  READY   STATUS    RESTARTS   AGE     IP          NODE
nginx-6fffd8d664-8t7rv   1/1     Running   0          2m30s   10.0.0.9    192.168.5.8
nginx-6fffd8d664-dpwbk   1/1     Running   0          5m12s   10.0.0.132   192.168.5.112
nginx-6fffd8d664-h796b   1/1     Running   0          78s     10.0.1.5    192.168.5.179
nginx-6fffd8d664-qhclc   1/1     Running   0          5m12s   10.0.1.133   192.168.5.252
```

5.3 Implementing High Availability for Add-ons in CCE

Application Scenarios

CCE offers various add-ons that enhance the cloud native capabilities of clusters. These add-ons include features like container scheduling and elasticity, cloud native observability, container networking, storage, and security. Helm charts are used to deploy these add-ons. Workload pods of the add-ons are deployed on worker nodes within the clusters.

As add-ons have become more popular, their stability and reliability have become essential requirements. By default, CCE implements a policy for add-on

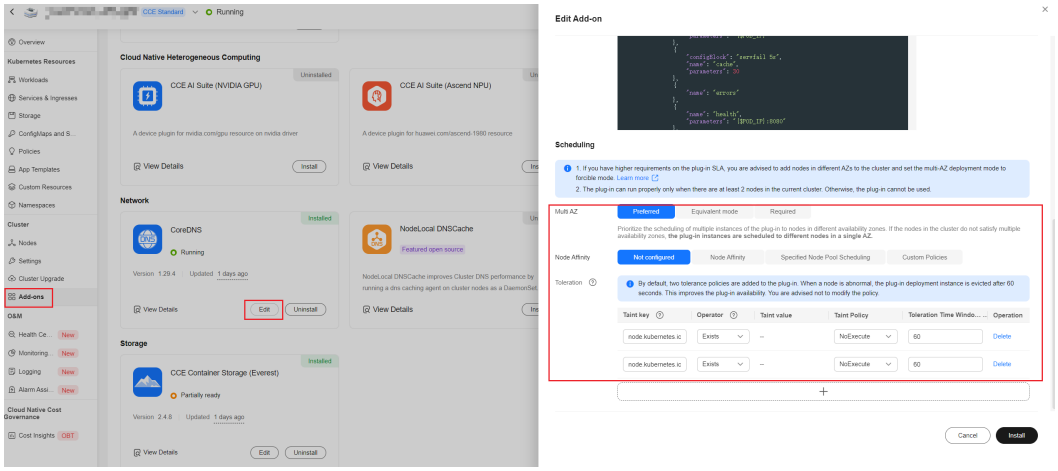
deployment where worker nodes have a hard anti-affinity configuration, and AZs have a soft anti-affinity configuration. This section explains how to enhance the CCE add-on scheduling policy, allowing you to customize the deployment policy according to your requirements.

Viewing the Scheduling Policy of an Add-on

An add-on typically includes Deployments and daemon processes. By default, daemon processes are deployed on all nodes, while Deployments are deployed in multi-instance mode for HA scenarios.

Take the CoreDNS add-on as an example. Two pods are deployed for this add-on by default, and multi-AZ deployment is in the preferred mode. The scheduling policy is hard anti-affinity for nodes and soft anti-affinity for AZs. As a result, to ensure that all pods run smoothly, the cluster requires two nodes. The Deployment pods of the add-on are scheduled to nodes in different AZs. However, if the nodes in the cluster are not in multiple AZs, the add-on pods will be scheduled to different nodes in a single AZ.

Figure 5-3 CoreDNS' scheduling policy



Creating Nodes in Different AZs for a Cluster

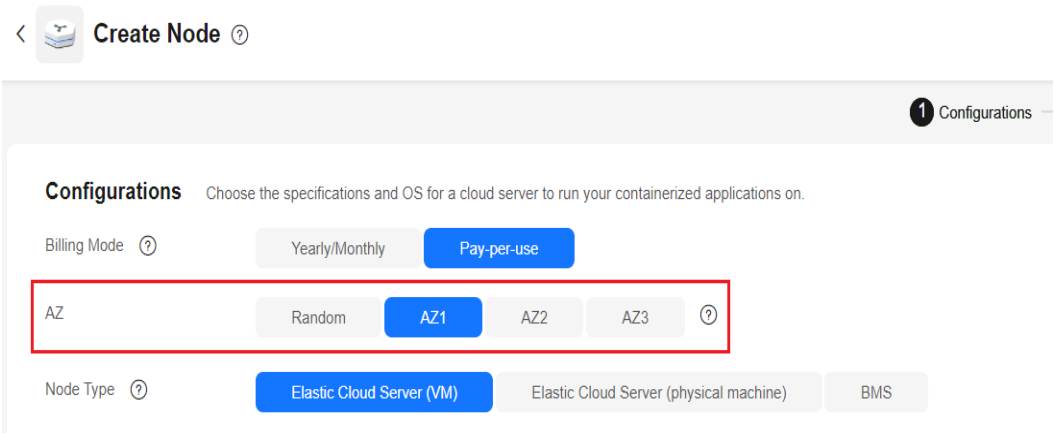
Step 1 Log in to the CCE console.

Step 2 In the navigation pane, choose **Clusters**. Click the target cluster name to access its details page.

Step 3 In the navigation pane, choose **Nodes**, click the **Nodes** tab, and click **Create Node** in the upper right corner.

Step 4 On the page displayed, select an AZ for the node.

Figure 5-4 Creating a node



Step 5 Configure other mandatory parameters following instructions to complete the creation.

----End

To create nodes in different AZs, you can simply repeat the previous steps. Alternatively, you can create multiple node pools, associate them with different AZ flavors, and increase the number of nodes in each pool to achieve the same result.

Configuring the AZ Anti-affinity Scheduling Policy for the Add-on

By default, the add-on scheduling policy can handle single-node faults. However, if your services require a higher SLA, you can create nodes with different AZ specifications on the node pool page. Then, set the multi-AZ deployment mode of the add-on scheduling policy to the required mode.

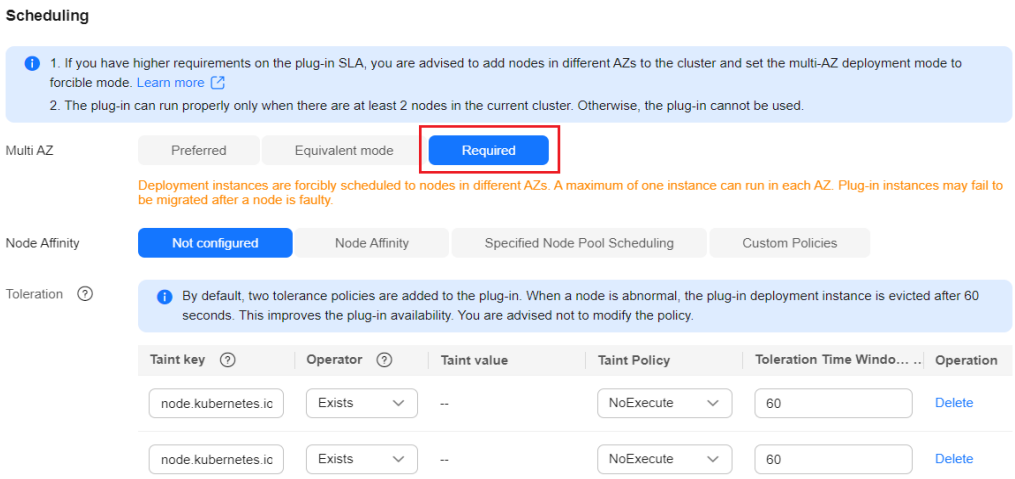
This section uses the CoreDNS add-on as an example to describe how to configure the multi-AZ deployment policy for an add-on.

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 In the navigation tree, choose **Add-ons**. In the right pane, locate **CoreDNS** and click **Edit**.

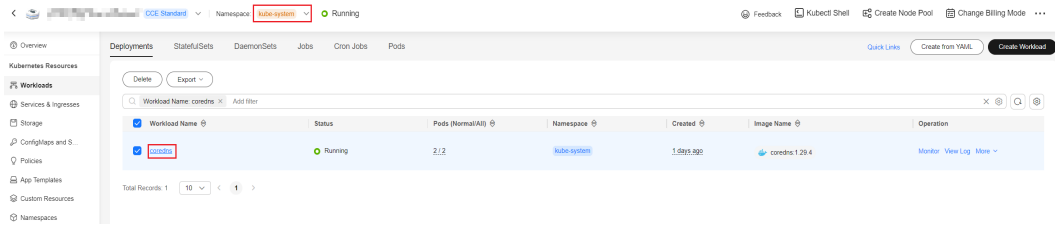
Step 3 In the window that slides out from the right, set **Multi AZ** to **Required** and click **Install**.

Figure 5-5 Changing the multi-AZ deployment mode to the required mode



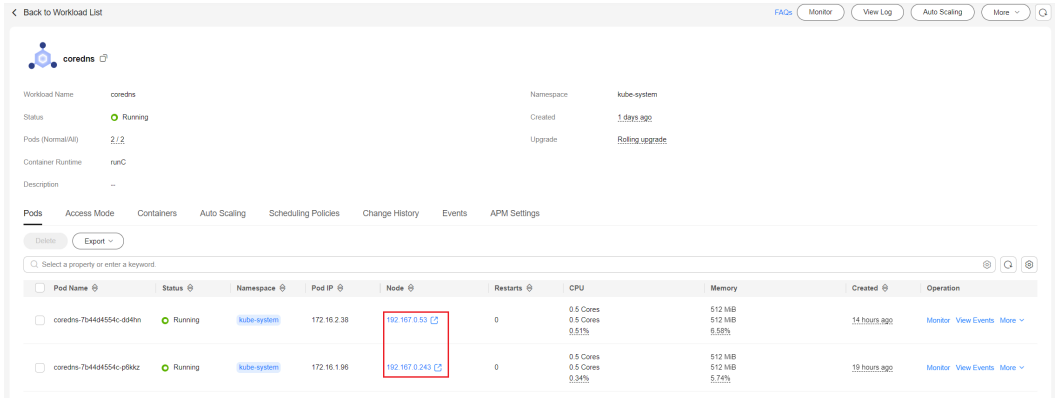
Step 4 Choose **Workload**, click the **Deployments** tab, and view the CoreDNS pods. Select the **kube-system** namespace to view the pod distribution of the add-on.

Figure 5-6 Viewing the deployment and distribution of CoreDNS pods



Step 5 View that the Deployment pods of the add-on has been allocated to nodes in two AZs.

Figure 5-7 Viewing CoreDNS pod distribution



----End

6 Security

6.1 Suggestions on Selecting CCE Clusters

Based on the shared security responsibility model, CCE safeguards the master nodes in a cluster and CCE components, and provides a series of hierarchical security capabilities at the cluster and container levels. Users are responsible for the security of cluster nodes and comply with the security best practices provided by CCE to perform security configuration and O&M.

CCE Application Scenarios

CCE is a container service built on popular Docker and Kubernetes technologies and offers a wealth of features best suited to enterprises' demand for running container clusters at scale. With unique advantages in system reliability, performance, and compatibility with open-source communities, CCE can suit the diverse needs of enterprises interested in building container clouds.

CCE provides a function list and typical application scenarios. For details about the function list, see **Function Overview**. For details about the application scenarios, see **Application Scenarios**.

Exception Scenarios

You are not advised to use clusters in scenarios that require strong resource isolation. CCE provides tenants with a dedicated, exclusive cluster. Currently, resources such as nodes and networks are not strictly isolated. If no strict security protection measures are available, security risks exist when the cluster is used by multiple external uncontrollable users at the same time. For example, in a development pipeline scenario, when multiple users are allowed to use the pipeline, the service code logic of different users is uncontrollable, and the cluster and services in the cluster may be attacked.

Enabling HSS

Host Security Service (HSS) provides host management, risk prevention, intrusion detection, advanced defense, security operations, and web page anti-tamper functions to comprehensively identify and manage information assets on hosts,

monitor risks on hosts in real time, and prevent unauthorized intrusions. You are advised to enable HSS to protect hosts in CCE clusters. For details about HSS and how to use it, see **HSS**.

Enabling CGS

CCE can be used together with Container Guard Service (CGS). CGS scans vulnerabilities and configurations in images, helping enterprises detect the container environment, which cannot be found by the traditional security software. CGS also delivers functions such as process whitelist configuration, read-only file protection, and container escape detection to minimize the security risks for a running container. For details about CGS and how to use it, see **CGS**.

6.2 Cluster Security

For security purposes, you are advised to configure a cluster as follows.

Using the CCE Cluster of the Latest Version

Kubernetes releases a major version in about four months. CCE follows the same frequency as Kubernetes to release major versions. To be specific, a new CCE version is released about three months after a new Kubernetes version is released in the community. For example, Kubernetes v1.19 was released in September 2020 and CCE v1.19 was released in March 2021.

The latest cluster version has known vulnerabilities fixed or provides a more comprehensive security protection mechanism. You are advised to select the latest cluster version when creating a cluster. Before a cluster version is deprecated and removed, upgrade your cluster to a supported version.

Handling Vulnerabilities Released on the Official Website Promptly

CCE releases vulnerabilities irregularly. You need to handle the vulnerabilities in a timely manner. For details, see **Vulnerability Notice**.

Disabling the Automatic Token Mounting Function of the Default Service Account

By default, Kubernetes associates the default service account with every pod, which means that the token is mounted to a container. The container can use this token to pass the authentication by the kube-apiserver and kubelet components. In a cluster with RBAC disabled, the service account who owns the token has the control permissions for the entire cluster. In a cluster with RBAC enabled, the permissions of the service account who owns the token depends on the roles associated by the administrator. The service account's token is generally used by workloads that need to access kube-apiserver, such as coredns, autoscaler, and prometheus. For workloads that do not need to access kube-apiserver, you are advised to disable the automatic association between the service account and token.

Two methods are available:

- Method 1: Set the **automountServiceAccountToken** field of the service account to **false**. After the configuration is complete, newly created workloads will not be associated with the default service account by default. Configure this field for each namespace as required.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
automountServiceAccountToken: false
...
```

  When a workload needs to be associated with a service account, explicitly set **automountServiceAccountToken** to **true** in the YAML file of the workload.

```
...
 spec:
   template:
     spec:
       serviceAccountName: default
       automountServiceAccountToken: true
       ...
```

- Method 2: Explicitly disable the function of automatically associating service accounts with workloads.

```
...
 spec:
   template:
     spec:
       automountServiceAccountToken: false
       ...
```

Configuring Proper Cluster Access Permissions for Users

CCE allows you to create multiple IAM users. Your account can create different user groups, assign different access permissions to different user groups, and add users to the user groups with corresponding permissions when creating IAM users. In this way, users can control permissions on different regions and assign read-only permissions. Your account can also assign namespace-level permissions for users or user groups. To ensure security, it is advised that minimum user access permissions are assigned.

If you need to create multiple IAM users, configure the permissions of the IAM users and namespaces properly.

- For details about how to configure cluster permissions, see **Cluster Permissions (IAM-based)**.
- For details about how to configure namespace permissions, see **Namespace Permissions (Kubernetes RBAC-based)**.

Configuring Resource Quotas for Cluster Namespaces

CCE provides resource quota management, which allows users to limit the total amount of resources that can be allocated to each namespace. These resources include CPU, memory, storage volumes, pods, Services, Deployments, and StatefulSets. Proper configuration can prevent excessive resources created in a namespace from affecting the stability of the entire cluster.

For details, see **Setting a Resource Quota**.

Configuring LimitRange for Containers in a Namespace

With resource quotas, cluster administrators can restrict the use and creation of resources by namespace. In a namespace, a pod or container can use the maximum CPU and memory resources defined by the resource quota of the namespace. In this case, a pod or container may monopolize all available resources in the namespace. You are advised to configure LimitRange to restrict resource allocation within the namespace. The LimitRange parameter has the following restrictions:

- Limits the minimum and maximum resource usage of each pod or container in a namespace.

  For example, create the maximum and minimum CPU usage limits for a pod in a namespace as follows:

  cpu-constraints.yaml

  ```
  apiVersion: v1
  kind: LimitRange
  metadata:
    name: cpu-min-max-demo-lr
  spec:
    limits:
    - max:
        cpu: "800m"
      min:
        cpu: "200m"
      type: Container
  ```

  Then, run **kubectl -n** *<namespace>* **create -f** *cpu-constraints.yaml* to complete the creation. If the default CPU usage is not specified for the container, the platform automatically configures the default CPU usage. That is, the default configuration is automatically added after the container is created.

  ```
  ...
  spec:
    limits:
    - default:
        cpu: 800m
      defaultRequest:
        cpu: 800m
      max:
        cpu: 800m
      min:
        cpu: 200m
      type: Container
  ```

- Limits the maximum and minimum storage space that each PersistentVolumeClaim can apply for in a namespace.

  storagelimit.yaml

  ```
  apiVersion: v1
  kind: LimitRange
  metadata:
    name: storagelimit
  spec:
    limits:
    - type: PersistentVolumeClaim
      max:
        storage: 2Gi
      min:
        storage: 1Gi
  ```

  Then, run **kubectl -n** *<namespace>* **create -f** *storagelimit.yaml* to complete the creation.

Configuring Network Isolation in a Cluster

- Container tunnel network

  If networks need to be isolated between namespaces in a cluster or between workloads in the same namespace, you can configure network policies to isolate the networks. For details, see **Network Policies**.

- Cloud Native Network 2.0

  In the Cloud Native Network 2.0 model, you can configure security groups to isolate networks between pods. For details, see **Security Group Policies**.

- VPC network

  Network isolation is not supported.

Enabling the Webhook Authentication Mode with kubelet

NOTICE

CCE clusters of v1.15.6-r1 or earlier are involved, whereas versions later than v1.15.6-r1 are not.

Upgrade the CCE cluster version to 1.13 or 1.15 and enable the RBAC capability for the cluster. If the version is 1.13 or later, no upgrade is required.

When creating a node, you can enable the kubelet authentication mode by injecting the **postinstall** file (by setting the kubelet startup parameter **--authorization-node=Webhook**).

Step 1 Run the following command to create clusterrolebinding:

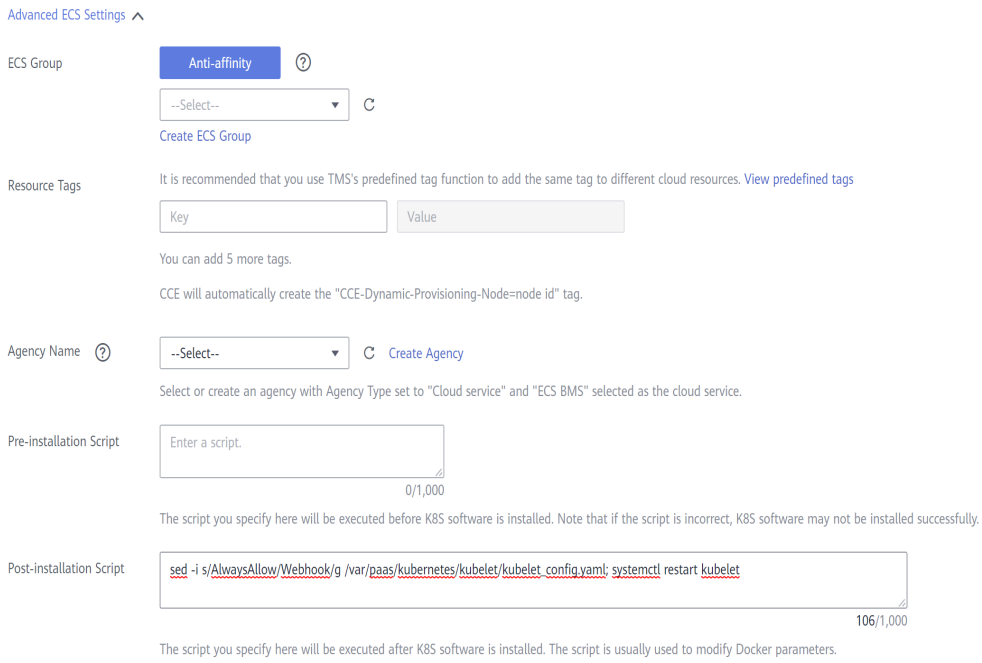
kubectl create clusterrolebinding kube-apiserver-kubelet-admin --clusterrole=system:kubelet-api-admin --user=system:kube-apiserver

Step 2 For an existing node, log in to the node, change **authorization mode** in **/var/paas/kubernetes/kubelet/kubelet_config.yaml** on the node to **Webhook**, and restart kubelet.

sed -i s/AlwaysAllow/Webhook/g /var/paas/kubernetes/kubelet/kubelet_config.yaml; systemctl restart kubelet

Step 3 For a new node, add the following command to the post-installation script to change the kubelet permission mode:

sed -i s/AlwaysAllow/Webhook/g /var/paas/kubernetes/kubelet/kubelet_config.yaml; systemctl restart kubelet



----End

Uninstalling web-terminal After Use

The web-terminal add-on can be used to manage CCE clusters. Keep the login password secure and uninstall the add-on when it is no longer needed.

6.3 Node Security

Handling Vulnerabilities Released on the Official Website Promptly

Before releasing a new image, fix the node vulnerabilities by referring to **Vulnerability Notice**.

Preventing Nodes from Being Exposed to Public Networks

- Do not bind an EIP to a node unless necessary to reduce the attack surface.
- If an EIP must be used, properly configure the firewall or security group rules to restrict access of unnecessary ports and IP addresses.

You may have configured the **kubeconfig.json** file on a node in your cluster. kubectl can use the certificate and private key in this file to control the entire cluster. You are advised to delete unnecessary files from the **/root/.kube** directory on the node to prevent malicious use.

rm -rf /root/.kube

Hardening VPC Security Group Rules

CCE is a universal container platform. Its default security group rules apply to common scenarios. Based on security requirements, you can harden the security

group rules set for CCE clusters on the **Security Groups** page of **Network Console**.

For details, see **Configuring Cluster Security Group Rules.**

Hardening Nodes on Demand

CCE cluster nodes use the default settings of open source OSs. After a node is created, you need to perform security hardening according to your service requirements.

In CCE, you can perform hardening as follows:

- Use the post-installation script after the node is created. For details, see the description about **Post-installation Script** in **Advanced Settings** when creating a node. This script is user-defined.
- Build custom images in CCE to create worker nodes. For details about the creation process, see **Creating a Custom CCE Node Image**.

Forbidding Containers to Obtain Host Machine Metadata

If a single CCE cluster is shared by multiple users to deploy containers, containers cannot access the management address (169.254.169.254) of OpenStack, preventing containers from obtaining metadata of host machines.

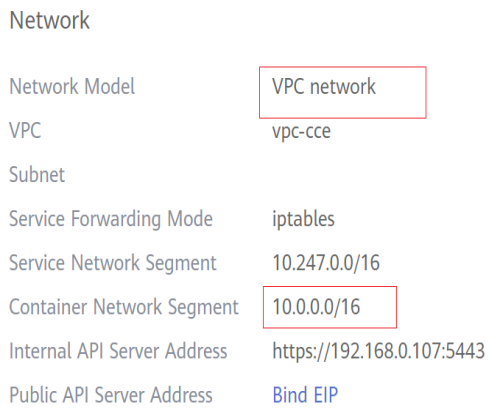
For details about how to restore the metadata, see the "Notes" section in **Obtaining Metadata**.

WARNING

This solution may affect the password change on the ECS console. Therefore, you must verify the solution before rectifying the fault.

Step 1 Obtain the network model and container CIDR of the cluster.

On the **Clusters** page of the CCE console, view the network model and container CIDR of the cluster.



Step 2 Prevent the container from obtaining host metadata.

- VPC network
  a. Log in to each node in the cluster as user **root** and run the following command:
     ```
     iptables -I OUTPUT -s {container_cidr} -d 169.254.169.254 -j REJECT
     ```
     {container_cidr} indicates the container CIDR of the cluster, for example, **10.0.0.0/16**.
     To ensure configuration persistence, write the command to the **/etc/rc.local** script.
  b. Run the following commands in the container to access the **userdata** and **metadata** interfaces of OpenStack and check whether the request is intercepted:
     ```
     curl 169.254.169.254/openstack/latest/meta_data.json
     curl 169.254.169.254/openstack/latest/user_data
     ```
- Container tunnel network
  a. Log in to each node in the cluster as user **root** and run the following command:
     ```
     iptables -I FORWARD -s {container_cidr} -d 169.254.169.254 -j REJECT
     ```
     {container_cidr} indicates the container CIDR of the cluster, for example, **10.0.0.0/16**.
     To ensure configuration persistence, write the command to the **/etc/rc.local** script.
  b. Run the following commands in the container to access the **userdata** and **metadata** interfaces of OpenStack and check whether the request is intercepted:
     ```
     curl 169.254.169.254/openstack/latest/meta_data.json
     curl 169.254.169.254/openstack/latest/user_data
     ```
- CCE Turbo cluster
  No additional configuration is required.

----End

6.4 Container Security

Controlling the Pod Scheduling Scope

The nodeSelector or nodeAffinity is used to limit the range of nodes to which applications can be scheduled, preventing the entire cluster from being threatened due to the exceptions of a single application. For details, see **Node Affinity**.

Suggestions on Container Security Configuration

- Set the computing resource limits (**request** and **limit**) of a container. This prevents the container from occupying too many resources and affecting the stability of the host and other containers on the same node.
- Unless necessary, do not mount sensitive host directories to containers, such as **/**, **/boot**, **/dev**, **/etc**, **/lib**, **/proc**, **/sys**, and **/usr**.
- Do not run the sshd process in containers unless necessary.
- Unless necessary, it is not recommended that containers and hosts share the network namespace.

- Unless necessary, it is not recommended that containers and hosts share the process namespace.
- Unless necessary, it is not recommended that containers and hosts share the IPC namespace.
- Unless necessary, it is not recommended that containers and hosts share the UTS namespace.
- Unless necessary, do not mount the sock file of Docker to any container.

Container Permission Access Control

When using a containerized application, comply with the minimum privilege principle and properly set securityContext of Deployments or StatefulSets.

- Configure runAsUser to specify a non-root user to run a container.
- Configure privileged to prevent containers being used in scenarios where privilege is not required.
- Configure capabilities to accurately control the privileged access permission of containers.
- Configure allowPrivilegeEscalation to disable privilege escape in scenarios where privilege escalation is not required for container processes.
- Configure seccomp to restrict the container syscalls. For details, see **Restrict a Container's Syscalls with seccomp** in the official Kubernetes documentation.
- Configure ReadOnlyRootFilesystem to protect the root file system of a container.

  Example YAML for a Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: security-context-example
  namespace: security-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: security-context-example
      label: security-context-example
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      annotations:
        seccomp.security.alpha.kubernetes.io/pod: runtime/default
      labels:
        app: security-context-example
        label: security-context-example
    spec:
      containers:
        - image: ...
          imagePullPolicy: Always
          name: security-context-example
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            runAsUser: 1000
            capabilities:
```



```
        add:
        - NET_BIND_SERVICE
        drop:
        - all
      volumeMounts:
        - mountPath: /etc/localtime
          name: localtime
          readOnly: true
        - mountPath: /opt/write-file-dir
          name: tmpfs-example-001
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  volumes:
    - hostPath:
        path: /etc/localtime
        type: ""
      name: localtime
    - emptyDir: {}
      name: tmpfs-example-001
```

Restricting the Access of Containers to the Management Plane

If application containers on a node do not need to access Kubernetes, you can perform the following operations to disable containers from accessing kube-apiserver:

Step 1 Query the container CIDR block and private API server address.

On the **Clusters** page of the CCE console, click the name of the cluster to find the information on the details page.

Step 2 Configure access rules.

- CCE cluster: Log in to each node in the cluster as user **root** and run the following command:
  - VPC network:
    ```
    iptables -I OUTPUT -s {container_cidr} -d {Private API server IP} -j REJECT
    ```
  - Container tunnel network:
    ```
    iptables -I FORWARD -s {container_cidr} -d {Private API server IP} -j REJECT
    ```

  {container_cidr} indicates the container CIDR of the cluster, for example, 10.0.0.0/16.

  To ensure configuration persistence, you are advised to write the command to the **/etc/rc.local** script.
- CCE Turbo cluster: Add an outbound rule to the ENI security group of the cluster.
  a. Log in to the VPC console.
  b. In the navigation pane, choose **Access Control** > **Security Groups**.
  c. Locate the ENI security group corresponding to the cluster and name it in the format of *{Cluster name}*-**cce-eni-***{Random ID}*. Click the security group name and configure rules.
  d. Click the **Outbound Rules** tab and click **Add Rule** to add an outbound rule for the security group.
     - **Priority**: Set it to **1**.

- **Action**: Select **Deny**, indicating that the access to the destination address is denied.
- **Type**: Select **IPv4**.
- **Protocol & Port**: Enter **5443** based on the port in the intranet API server address.
- **Destination**: Select **IP address** and enter the IP address of the internal API server.

e. Click **OK**.

Step 3 Run the following command in the container to access kube-apiserver and check whether the request is intercepted:

```
curl -k https://{Private API server IP}:5443
```

----End

6.5 Secret Security

Currently, CCE has configured static encryption for secret resources. The secrets created by users will be encrypted and stored in etcd of the CCE cluster. Secrets can be used in two modes: environment variable and file mounting. No matter which mode is used, CCE still transfers the configured data to users. Therefore, it is recommended that:

1. Do not record sensitive information in logs.
2. For the secret that uses the file mounting mode, the default file permission mapped in the container is 0644. Configure stricter permissions for the file. For example:

```
apiversion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      defaultMode: 256
```

   In **defaultMode: 256**, **256** is a decimal number, which corresponds to the octal number **0400**.

3. When the file mounting mode is used, configure the secret file name to hide the file in the container.

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
apiVersion: v1
```



```
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
  - name: secret-volume
    secret:
      secretName: dotfile-secret
  containers:
  - name: dotfile-test-container
    image: k8s.gcr.io/busybox
    command:
    - ls
    - "-1"
    - "/etc/secret-volume"
    volumeMounts:
    - name: secret-volume
      readOnly: true
      mountPath: "/etc/secret-volume"
```

In this way, **.secret-file** cannot be viewed by running the **ls -l** command in the **/etc/secret-volume/** directory, but can be viewed by running the **ls -al** command.

4. Encrypt sensitive information before creating a secret and decrypt the information when using it.

Using a Bound ServiceAccount Token to Access a Cluster

The secret-based ServiceAccount token does not support expiration time or auto update. In addition, after the mounting pod is deleted, the token is still stored in the secret. Token leakage may incur security risks. A bound ServiceAccount token is recommended for CCE clusters of version 1.23 or later. In this mode, the expiration time can be set and is the same as the pod lifecycle, reducing token leakage risks. Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: security-token-example
  namespace: security-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: security-token-example
      label: security-token-example
  template:
    metadata:
      annotations:
        seccomp.security.alpha.kubernetes.io/pod: runtime/default
      labels:
        app: security-token-example
        label: security-token-example
    spec:
      serviceAccountName: test-sa
      containers:
        - image: ...
          imagePullPolicy: Always
          name: security-token-example
      volumes:
        - name: test-projected
          projected:
            defaultMode: 420
            sources:
              - serviceAccountToken:
                  expirationSeconds: 1800
```



```
      path: token
  - configMap:
      items:
        - key: ca.crt
          path: ca.crt
      name: kube-root-ca.crt
  - downwardAPI:
      items:
        - fieldRef:
            apiVersion: v1
            fieldPath: metadata.namespace
          path: namespace
```

For details, visit https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/.

6.6 Workload Identities

With workload identities, your workloads in a cluster can access cloud services like IAM without using the AK/SK, reducing security risks.

This section describes how to use workload identities in CCE.

Notes and Constraints

Your clusters must be version 1.19.16 or later.

Procedure

1. Obtain the JSON Web Key Set (JWKS) of the cluster (the signature public key of the service account token) from CCE.
2. Create an identity provider on the IAM console.
3. Deploy the application and bind it with the identity provider.
   a. Use the OIDC token to access IAM and obtain the IAM token (implemented by you).
   b. Use the IAM token to access cloud services (implemented by you).

Obtaining JWKS of a CCE Cluster

Step 1 Use kubectl to connect to the cluster.

Step 2 Run the following command to obtain the public key:

kubectl get --raw /openid/v1/jwks

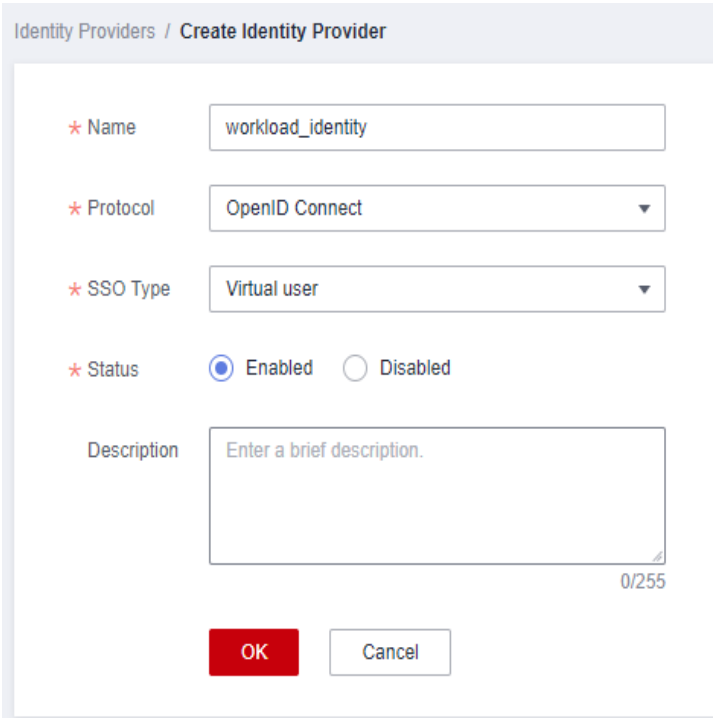
```
# kubectl get --raw /openid/v1/jwks
{"keys":[{"use":"sig","kty":"RSA","kid":"*****","alg":"RS256","n":"*****","e":"AQAB"}]}
```

The returned field is the public key of the cluster.

----End

Configuring an Identity Provider

Step 1 Log in to the IAM console, create an identity provider, and select **OpenID Connect** for **Protocol**.

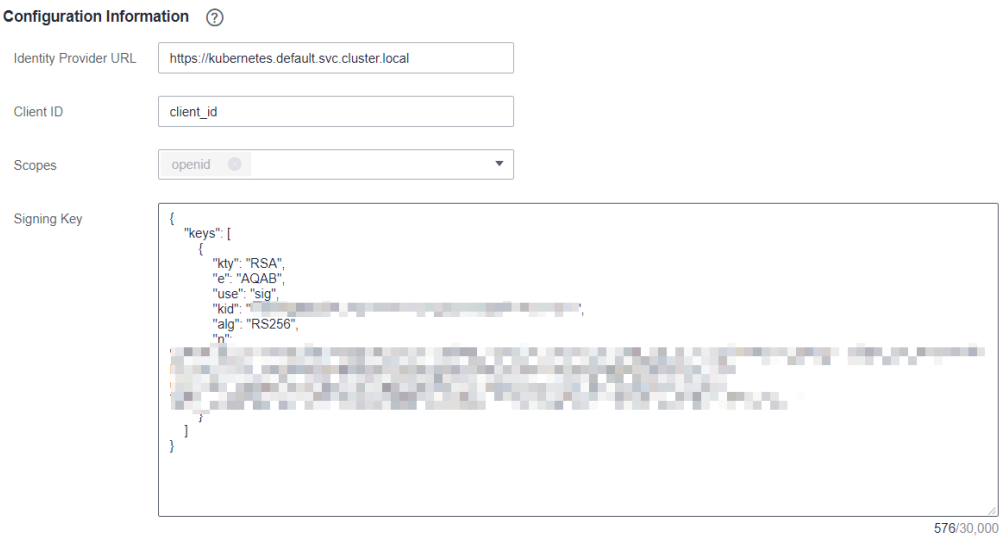


Step 2 Click **OK**. After the creation, modify the identity provider information.

Access Type: Select **Programmatic access**.

Configuration Information

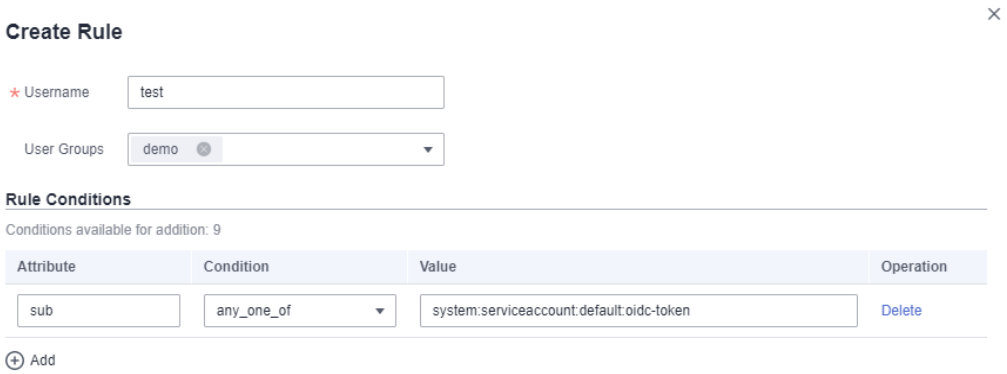
- **Identity Provider URL**: Enter **https://kubernetes.default.svc.cluster.local**.
- **Client ID**: Enter an ID, which will be used when you create a container.
- **Signing Key**: Enter the JWKS of the CCE cluster obtained in **Obtaining JWKS of a CCE Cluster**.



Identity Conversion Rules

An identity conversion rule maps the ServiceAccount of a workload to an IAM user group.

For example, create a ServiceAccount named **oidc-token** in namespace **default** of the cluster and map it to user group **demo**. If you use the identity provider ID to access cloud services, you have the permissions of the **demo** user group. The attribute must be **sub**. The value format is **system:serviceaccount:*Namespace*:*ServiceAccountName***.



Rules are in the JSON format as follows:

```
[
  {
    "local": [
      {
        "user": {
          "name": "test"
        }
      },
      {
        "group": {
          "name": "demo"
        }
      }
    ],
    "remote": [
      {
        "type": "sub",
        "any_one_of": [
          "system:serviceaccount:default:oidc-token"
        ]
      }
    ]
  }
]
```

Step 3 Click **OK**.

----End

Using a Workload Identity

Create a ServiceAccount, whose name must be the value of **ServiceAccountName** set in **Configuring an Identity Provider**.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: oidc-token
```

Example configuration for the workload:

```
apiVersion: apps/v1
kind: Deployment
```



```
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:
        app: nginx
        version: v1
    spec:
      containers:
      - name: container-1
        image: nginx:latest
        volumeMounts:
        - mountPath: "/var/run/secrets/tokens"     # Mount the serviceAccountToken generated by Kubernetes to the /var/run/secrets/tokens/oidc-token file.
          name: oidc-token
      imagePullSecrets:
      - name: default-secret
      serviceAccountName: oidc-token      # Name of the created ServiceAccount
      volumes:
      - name: oidc-token
        projected:
          defaultMode: 420
          sources:
          - serviceAccountToken:
              audience: client_id   # Must be the client ID of the identity provider.
              expirationSeconds: 7200       # Expiry period
              path: oidc-token              # Path name, which can be customized.
```

After the creation, log in to the container. The content of the **/var/run/secrets/tokens/oidc-token** file is the serviceAccountToken generated by Kubernetes. You can obtain the IAM token by calling the API for **obtaining a token with an OpenID Connect ID token**. The **X-Subject-Token** field in the response header is the IAM token. In this way, you can access cloud services using the IAM token.

NOTE

If the serviceAccountToken is used for over 24 hours or 80% of the expiry period, kubelet automatically rotates the serviceAccountToken.

An example is as follows:

```
curl -i --location --request POST 'https://{{iam endpoint}}/v3.0/OS-AUTH/id-token/tokens' \
 --header 'X-Idp-Id: workload_identity' \
 --header 'Content-Type: application/json' \
 --data @token_body.json
```

Specifically:

- **{{iam endpoint}}** indicates the endpoint of IAM. For details, see **Regions and Endpoints**.
- **workload_identity** is the identity provider name, which is the same as that configured in **Configuring an Identity Provider**.
- **token_body.json** is a local file and its content is as follows:

  ```
  {
    "auth" : {
      "id_token" : {
        "id" : "eyJhbGciOiJSU..."
      },
      "scope": {
  ```



```
    "project" : {
      "id" : "46419baef4324...",
      "name" : "cn-north-4"
    }
   }
  }
}
```

- **$.auth.id_token.id**: The value is the content of the **/var/run/secrets/tokens/oidc-token** file in the container.
- **$.auth.scope.project.id**: indicates the project ID. For details about how to obtain the project ID, see **Obtaining a Project ID**.
- **$.auth.scope.project.name**: indicates the project name, for example, **cn-north-4**.

7 Auto Scaling

7.1 Using HPA and CA for Auto Scaling of Workloads and Nodes

Application Scenarios

The best way to handle surging traffic is to automatically adjust the number of machines based on the traffic volume or resource usage, which is called scaling.

When pods or containers are used for deploying applications, the upper limit of available resources is typically required to set for pods or containers to prevent unlimited usage of node resources during peak hours. However, after the upper limit is reached, an application error may occur. To resolve this issue, scale in the number of pods to share workloads. If the node resource usage increases to a certain extent that newly added pods cannot be scheduled, scale in the number of nodes based on the node resource usage.

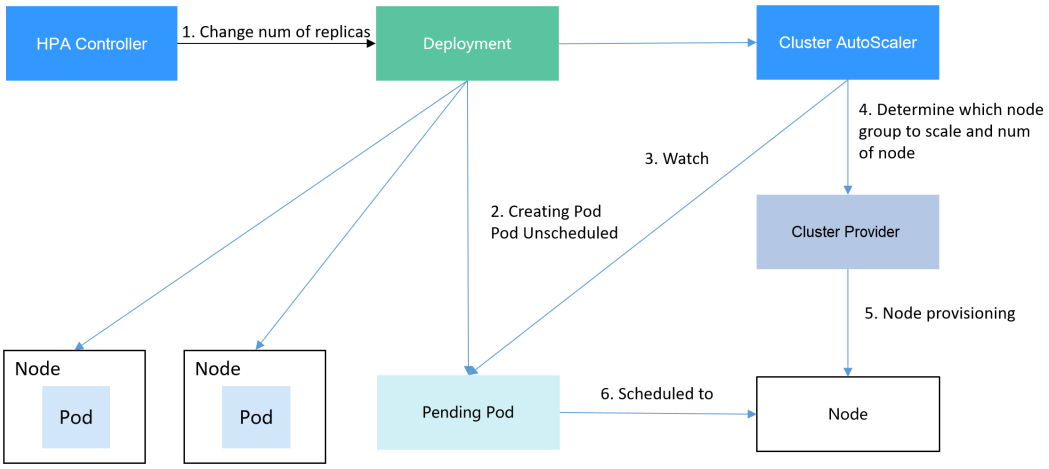
Solution

Two major auto scaling policies are HPA (Horizontal Pod Autoscaling) and CA (Cluster AutoScaling). HPA is for workload auto scaling and CA is for node auto scaling.

HPA and CA work with each other. HPA requires sufficient cluster resources for successful scaling. When the cluster resources are insufficient, CA is needed to add nodes. If HPA reduces workloads, the cluster will have a large number of idle resources. In this case, CA needs to release nodes to avoid resource waste.

As shown in **Figure 7-1**, HPA performs scale-out based on the monitoring metrics. When cluster resources are insufficient, newly created pods are in Pending state. CA then checks these pending pods and selects the most appropriate node pool based on the configured scaling policy to scale out the node pool. For details about how HPA and CA work, see **Workload Scaling Mechanisms** and **Node Scaling Mechanisms**.

Figure 7-1 HPA and CA working flows

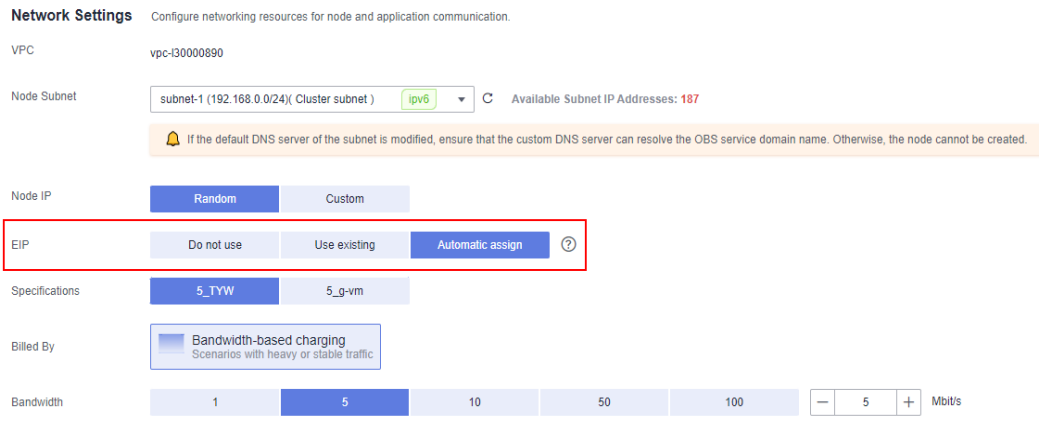


Using HPA and CA can easily implement auto scaling in most scenarios. In addition, the scaling process of nodes and pods can be easily observed.

This section uses an example to describe the auto scaling process using HPA and CA policies together.

Preparations

Step 1 Create a cluster with one node. The node should have 2 cores of vCPUs and 4 GiB of memory, or a higher specification, as well as an EIP to allow external access. If no EIP is bound to the node during node creation, you can manually bind one on the ECS console after creating the node.



Step 2 Install add-ons for the cluster.

- autoscaler: node scaling add-on
- metrics-server: an aggregator of resource usage data in a Kubernetes cluster. It can collect measurement data of major Kubernetes resources, such as pods, nodes, containers, and Services.

Step 3 Log in to the cluster node and run a computing-intensive application. When a user sends a request, the result needs to be calculated before being returned to the user.

1. Create a PHP file named **index.php** to calculate the square root of the request for 1,000,000 times before returning **OK!**.
   ```
   vi index.php
   ```
   The file content is as follows:
   ```
   <?php
    $x = 0.0001;
    for ($i = 0; $i <= 1000000; $i++) {
      $x += sqrt($x);
    }
    echo "OK!";
   ?>
   ```
2. Compile a **Dockerfile** file to build an image.
   ```
   vi Dockerfile
   ```
   The content is as follows:
   ```
   FROM php:5-apache
   COPY index.php /var/www/html/index.php
   RUN chmod a+rx index.php
   ```
3. Run the following command to build an image named **hpa-example** with the tag **latest**.
   ```
   docker build -t hpa-example:latest .
   ```
4. (Optional) Log in to the SWR console, choose **Organizations** in the navigation pane, and click **Create Organization** in the upper right corner to create an organization.

   Skip this step if you already have an organization.
5. In the navigation pane, choose **My Images** and then click **Upload Through Client**. On the page displayed, click **Generate a temporary login command** and click  to copy the command.
6. Run the login command copied in the previous step on the cluster node. If the login is successful, the message "Login Succeeded" is displayed.
7. Tag the hpa-example image.

   docker tag *{Image name 1:Tag 1}|{Image repository address}|{Organization name}|{Image name 2:Tag 2}*

   - *{Image name 1:Tag 1}*: name and tag of the local image to be uploaded.
   - *{Image repository address}*: the domain name at the end of the login command in **login command**. It can be obtained on the SWR console.
   - *{Organization name}*: name of the **created organization**.
   - *{Image name 2:Tag 2}*: desired image name and tag to be displayed on the SWR console.

   The following is an example:

   docker tag hpa-example:latest swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/hpa-example:latest
8. Push the image to the image repository.

   docker push *{Image repository address}|{Organization name}|{Image name 2:Tag 2}*

   The following is an example:

   docker push swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/hpa-example:latest

   The following information will be returned upon a successful push:
   ```
   6d6b9812c8ae: Pushed
   ...
   ```



```
fe4c16cbf7a4: Pushed
latest: digest: sha256:eb7e3bbd*** size: **
```

To view the pushed image, go to the SWR console and refresh the **My Images** page.

----End

Creating a Node Pool and a Node Scaling Policy

Step 1 Log in to the CCE console, access the created cluster, click **Nodes** on the left, click the **Node Pools** tab, and click **Create Node Pool** in the upper right corner.

Step 2 Configure the node pool.

- **Nodes**: Set it to **1**, indicating that one node is created by default when a node pool is created.
- **Specifications**: 2 vCPUs | 4 GiB

Retain the defaults for other parameters. For details, see **Creating a Node Pool**.

Step 3 Locate the row containing the newly created node pool and click **Auto Scaling** in the upper right corner. For details, see **Creating a Node Scaling Policy**.

If the CCE Cluster Autoscaler add-on is not installed in the cluster, install it first. For details, see **CCE Cluster Autoscaler**.

- **Automatic scale-out**: If this function is enabled, nodes in a node pool will be automatically added based on the cluster load.
- **Customized Rule**: Click **Add Rule**. In the dialog box displayed, configure parameters. If the CPU allocation rate is greater than 70%, a node is added to each associated node pool. A node scaling policy needs to be associated with a node pool. Multiple node pools can be associated. When you need to scale nodes, node with proper specifications will be added or reduced from the node pool based on the minimum waste principle.
- **Automatic scale-in**: If this function is enabled, nodes in a node pool will be automatically deleted based on the cluster load. For example, trigger scale-in when the node resource utilization is less than 50%.
- **AS Configuration**: Modify the node quantity range. During autoscaling, the number of nodes in a node pool is always within the configured quantity range.
- **AS Object**: Enable autoscaling for node specifications in a node pool.

Step 4 Click **OK**.

----End

Creating a Workload

Use the hpa-example image to create a Deployment with one replica. The image path is related to the organization uploaded to the SWR repository and needs to be replaced with the actual value.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hpa-example
spec:
```



```
  replicas: 1
  selector:
    matchLabels:
      app: hpa-example
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
      - name: container-1
        image: 'hpa-example:latest' # Replace it with the address of the image you uploaded to SWR.
        resources:
          limits:                   # The value of limits must be the same as that of requests to prevent flapping during scaling.
            cpu: 500m
            memory: 200Mi
          requests:
            cpu: 500m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

Then, create a NodePort Service for the workload so that the workload can be accessed from external networks.

NOTE

To allow external access to NodePort Services, allocate an EIP for the node in the cluster. After the allocation, synchronize node data. For details, see **Synchronizing Data with Cloud Servers**. If the node has already bound with an EIP, you do not need to create one.

Alternatively, you can create a Service with an ELB load balancer for external access. For details, see **Using kubectl to Create a Service (Automatically Creating a Shared Load Balancer)**.

```
kind: Service
apiVersion: v1
metadata:
  name: hpa-example
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 31144
  selector:
    app: hpa-example
  type: NodePort
```

Creating an HPA Policy

Create an HPA policy. As shown below, the policy is associated with the hpa-example workload, and the target CPU usage is 50%.

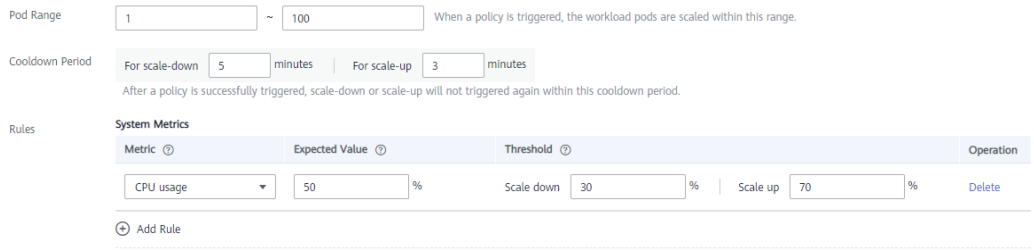
There are two other annotations. One annotation defines the CPU thresholds, indicating that scaling is not performed when the CPU usage is between 30% and 70% to prevent impact caused by slight fluctuation. The other is the scaling time window, indicating that after the policy is successfully executed, a scaling operation will not be triggered again in this cooling interval to prevent impact caused by short-term fluctuation.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
```



```
  name: hpa-policy
  annotations:
    extendedhpa.metrics: '[{"type":"Resource","name":"cpu","targetType":"Utilization","targetRange":{"low":"30","high":"70"}}]'
    extendedhpa.option: '{"downscaleWindow":"5m","upscaleWindow":"3m"}'
spec:
  scaleTargetRef:
    kind: Deployment
    name: hpa-example
    apiVersion: apps/v1
  minReplicas: 1
  maxReplicas: 100
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Configure the parameters as follows if you are using the console.



Observing the Auto Scaling Process

Step 1 Check the cluster node status. In the following example, there are two nodes.

```
# kubectl get node
NAME          STATUS   ROLES    AGE     VERSION
192.168.0.183   Ready    <none>   2m20s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   55m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

Check the HPA policy. The CPU usage of the target workload is 0%.

```
# kubectl get hpa hpa-policy
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%    1         100       1          4m
```

Step 2 Run the following command to access the workload. In the following command, {ip:port} indicates the access address of the workload, which can be queried on the workload details page.

while true;do wget -q -O- http://*{ip:port}*; done

NOTE

If no EIP is displayed, the cluster node has not been assigned any EIP. Allocate one, bind it to the node, and synchronize node data. For details, see **Synchronizing Data with Cloud Servers**.

Observe the scaling process of the workload.

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   0%/50%    1         100       1          4m
```



```
hpa-policy   Deployment/hpa-example   190%/50%   1   100   1   4m23s
hpa-policy   Deployment/hpa-example   190%/50%   1   100   4   4m31s
hpa-policy   Deployment/hpa-example   200%/50%   1   100   4   5m16s
hpa-policy   Deployment/hpa-example   200%/50%   1   100   4   6m16s
hpa-policy   Deployment/hpa-example   85%/50%    1   100   4   7m16s
hpa-policy   Deployment/hpa-example   81%/50%    1   100   4   8m16s
hpa-policy   Deployment/hpa-example   81%/50%    1   100   7   8m31s
hpa-policy   Deployment/hpa-example   57%/50%    1   100   7   9m16s
hpa-policy   Deployment/hpa-example   51%/50%    1   100   7   10m
hpa-policy   Deployment/hpa-example   58%/50%    1   100   7   11m
```

You can see that the CPU usage of the workload is 190% at 4m23s, which exceeds the target value. In this case, scaling is triggered to expand the workload to four replicas/pods. In the subsequent several minutes, the CPU usage does not decrease until 7m16s. This is because the new pods may not be successfully created. The possible cause is that resources are insufficient and the pods are in Pending state. During this period, nodes are added.

At 7m16s, the CPU usage decreases, indicating that the pods are successfully created and start to bear traffic. The CPU usage decreases to 81% at 8m, still greater than the target value (50%) and the high threshold (70%). Therefore, 7 pods are added at 9m16s, and the CPU usage decreases to 51%, which is within the range of 30% to 70%. From then on, the number of pods remains 7.

In the following output, you can see the workload scaling process and the time when the HPA policy takes effect.

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 7
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization
(percentage of request) above target
```

Check the number of nodes. The following output shows that two nodes are added.

```
# kubectl get node
NAME          STATUS   ROLES    AGE     VERSION
192.168.0.120   Ready    <none>   3m5s    v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.136   Ready    <none>   6m58s   v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.183   Ready    <none>   18m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
192.168.0.26    Ready    <none>   71m     v1.17.9-r0-CCE21.1.1.3.B001-17.36.8
```

You can also view the scaling history on the console. For example, the CA policy is executed once when the CPU allocation rate in the cluster is greater than 70%, and the number of nodes in the node pool is increased from 2 to 3. The new node is automatically added by autoscaler based on the pending state of pods in the initial phase of HPA.

The node scaling process is as follows:

1. After the number of pods changes to 4, the pods are in Pending state due to insufficient resources. As a result, the default scale-out policy of the autoscaler add-on is triggered, and the number of nodes is increased by one.
2. The second node scale-out is triggered because the CPU allocation rate in the cluster is greater than 70%. As a result, the number of nodes is increased by one, which is recorded in the scaling history on the console. Scaling based on the allocation rate ensures that the cluster has sufficient resources.

Step 3 Stop accessing the workload and check the number of pods.

```
# kubectl get hpa hpa-policy --watch
NAME         REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
hpa-policy   Deployment/hpa-example   50%/50%    1         100       7          12m
hpa-policy   Deployment/hpa-example   21%/50%    1         100       7          13m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          14m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       7          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          18m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          19m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       3          23m
hpa-policy   Deployment/hpa-example   0%/50%     1         100       1          23m
```

You can see that the CPU usage is 21% at 13m. The number of pods is reduced to 3 at 18m, and then reduced to 1 at 23m.

In the following output, you can see the workload scaling process and the time when the HPA policy takes effect.

```
# kubectl describe deploy hpa-example
...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  25m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 1
  Normal  ScalingReplicaSet  20m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 4
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up replica set hpa-example-79dd795485
to 7
  Normal  ScalingReplicaSet  6m28s  deployment-controller  Scaled down replica set hpa-
example-79dd795485 to 3
  Normal  ScalingReplicaSet  72s    deployment-controller  Scaled down replica set hpa-
example-79dd795485 to 1
# kubectl describe hpa hpa-policy
...
Events:
  Type    Reason             Age    From                       Message
  ----    ------             ----   ----                       -------
  Normal  SuccessfulRescale  20m    horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  16m    horizontal-pod-autoscaler  New size: 7; reason: cpu resource utilization
(percentage of request) above target
  Normal  SuccessfulRescale  6m45s  horizontal-pod-autoscaler  New size: 3; reason: All metrics below target
  Normal  SuccessfulRescale  90s    horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

You can also view the HPA policy execution history on the console. Wait until the one node is reduced.

The reason why the other two nodes in the node pool are not reduced is that they both have pods in the kube-system namespace (and these pods are not created by DaemonSets). For details, see **Node Scaling Mechanisms**.

----End

Summary

Using HPA and CA can easily implement auto scaling in most scenarios. In addition, the scaling process of nodes and pods can be easily observed.

7.2 Elastic Scaling of CCE Pods to CCI

CCE Cloud Bursting Engine for CCI functions as a virtual kubelet to connect Kubernetes clusters to APIs of other platforms. This add-on is mainly used to extend Kubernetes APIs to serverless container services such as Huawei Cloud CCI.

With this add-on, you can scale Deployments, StatefulSets, Jobs, and CronJobs running in CCE clusters to **Cloud Container Instance (CCI)** during peak hours. In this way, you can reduce consumption caused by cluster scaling.

Installing the Add-on

1. Log in to the CCE console.
2. Click the name of the target CCE cluster to go to the cluster console.
3. In the navigation pane on the left, choose **Add-ons**.
4. Select the **CCE Cloud Bursting Engine for CCI** add-on and click **Install**.
5. Configure the add-on parameters.

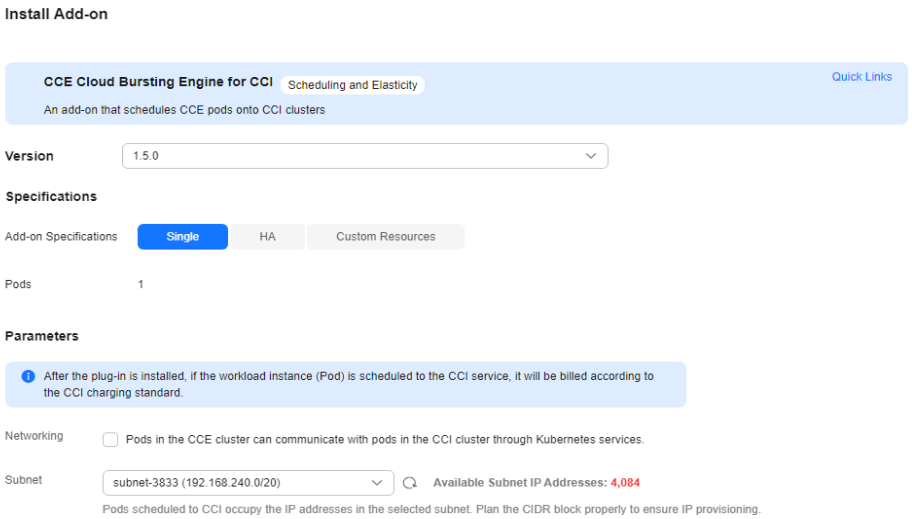


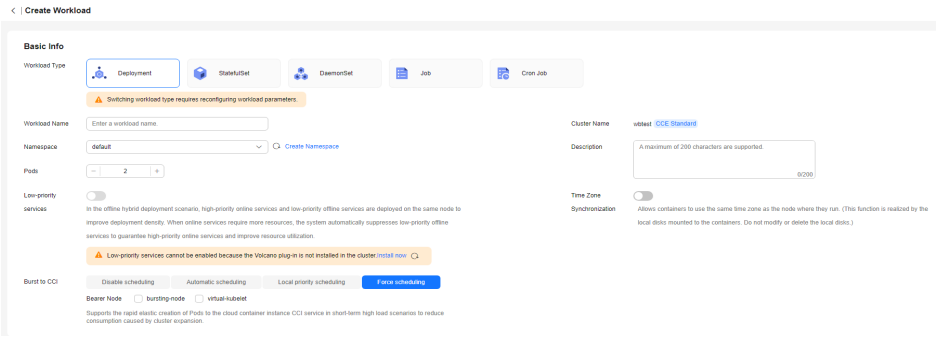
Table 7-1 Add-on parameters

Parameter	Description
Version	Add-on version. There is a mapping between add-on versions and CCE cluster versions. For more details, see "Change History" in **CCE Cloud Bursting Engine for CCI**.

Parameter	Description
Specifications	Number of pods required for a workload.
Networking	If this option is enabled, pods in a CCE cluster can communicate with the pods in CCI. For details, see **Networking**.

Creating a Workload

1. Log in to the CCE console.
2. Click the name of the target CCE cluster to go to the cluster console.
3. In the navigation pane on the left, choose **Workloads**.
4. Click **Create Workload**. For details, see **Creating a Workload**.
5. Specify basic information. Set **Burst to CCI** to **Force scheduling**. For more information about scheduling policies, see **Scaling Pods to CCI**.



6. Configure the container parameters.
7. Click **Create Workload**.
8. On the **Workloads** page, click the name of the created workload to go to the workload details page.
9. View the node where the workload is running. If the workload is running on a CCI node, it has been scheduled to CCI.

Uninstalling the Add-on

1. Log in to the CCE console.
2. Click the name of the target CCE cluster to go to the cluster console.
3. In the navigation pane on the left, choose **Add-ons**.
4. Select the **CCE Cloud Bursting Engine for CCI** add-on and click **Uninstall**.

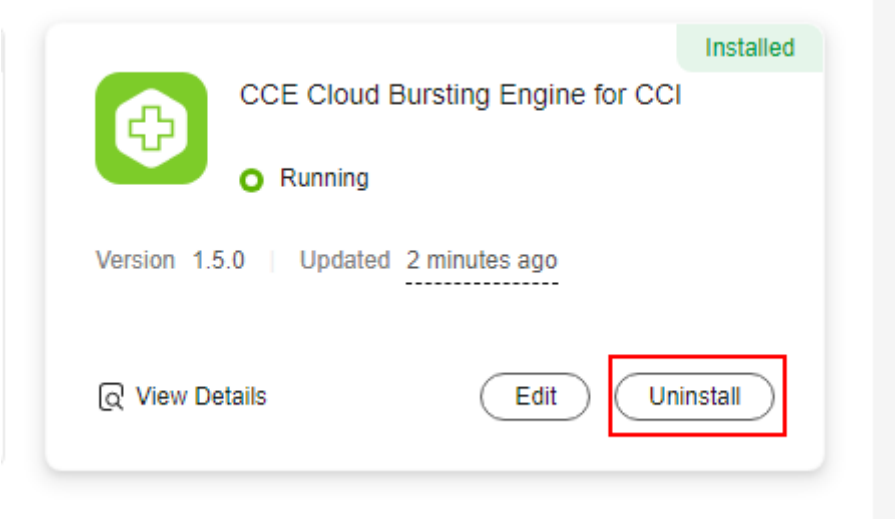


Table 7-2 Special scenarios for uninstalling the add-on

Scenario	Symptom	Description
There are no nodes in the CCE cluster that the bursting add-on needs to be uninstalled from.	Failed to uninstall the bursting add-on.	If the bursting add-on is uninstalled from the cluster, a job for clearing resources will be started in the cluster. To ensure that the job can be started, there is at least one node in the cluster that can be scheduled.
The CCE cluster is deleted, but the bursting add-on is not uninstalled.	There are residual resources in the namespace on CCI. If the resources are not free, additional expenditures will be generated.	The cluster is deleted, but the resource clearing job is not executed. You can manually clear the namespace and residual resources.

For more information about the bursting add-on, see **CCE Cloud Bursting Engine for CCI**.

7.3 Auto Scaling Based on ELB Monitoring Metrics

Background

By default, Kubernetes scales a workload based on resource usage metrics such as CPU and memory. However, this mechanism cannot reflect the real-time resource usage when traffic bursts arrive, because the collected CPU and memory usage data lags behind the actual load balancer traffic metrics. For some services (such as flash sale and social media) that require fast auto scaling, scaling based on this

rule may not be performed in a timely manner and cannot meet these services' actual needs. In this case, auto scaling based on ELB QPS data can respond to service requirements more timely.

Solution

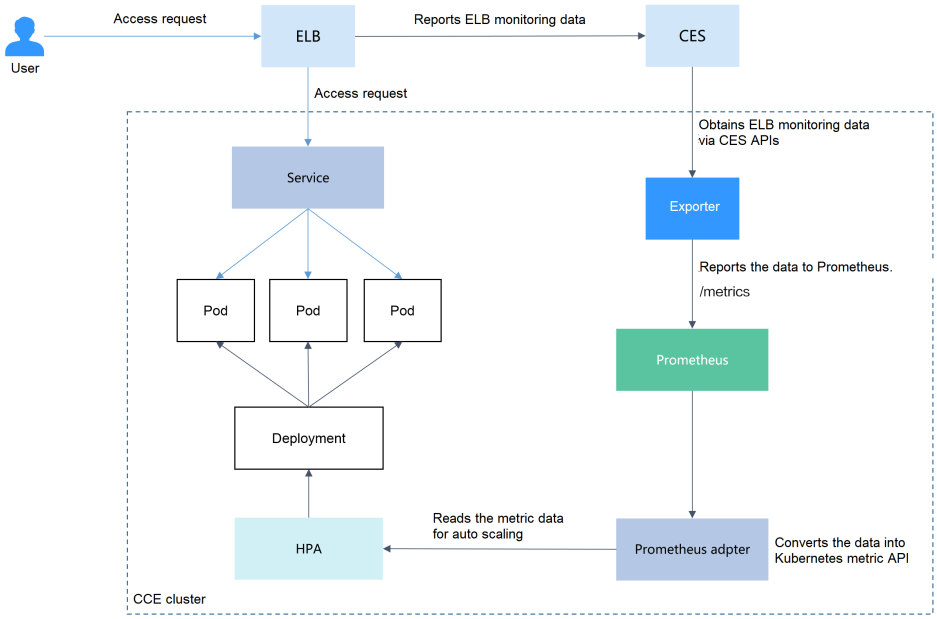
This section describes an auto scaling solution based on ELB monitoring metrics. Compared with CPU/memory usage-based auto scaling, auto scaling based on ELB QPS data is more targeted and timely.

The key of this solution is to obtain the ELB metric data and report the data to Prometheus, convert the data in Prometheus to the metric data that can be identified by HPA, and then perform auto scaling based on the converted data.

The implementation scheme is as follows:

1. Develop a Prometheus exporter to obtain ELB metric data, convert the data into the format required by Prometheus, and report it to Prometheus. This section uses **cloudeye-exporter** as an example.
2. Convert the Prometheus data into the Kubernetes metric API for the HPA controller to use.
3. Set an HPA rule to use ELB monitoring data as auto scaling metrics.

Figure 7-2 ELB traffic flows and monitoring data



NOTE

Other metrics can be collected in the similar way.

Prerequisites

- You must be familiar with Prometheus and be able to write the Prometheus exporter.
- You have the Cloud Native Cluster Monitoring add-on installed in your cluster. This add-on supports clusters of v1.17 or later.

> **NOTE**
>
> Set the deployment mode of Cloud Native Cluster Monitoring to the server mode.

Building an Exporter Image

This section uses **cloudeye-exporter** to monitor load balancer metrics. To develop an exporter, see **Appendix: Developing an Exporter**.

Step 1 Log in to a cluster node that can access the public network and compile a Dockerfile.

```
vi Dockerfile
```

The content is as follows:

```
FROM ubuntu:18.04
RUN apt-get update \
  && apt-get install -y git ca-certificates curl \
    && update-ca-certificates \
      && curl -O  https://dl.google.com/go/go1.14.14.linux-amd64.tar.gz \
        && tar -zxf go1.14.14.linux-amd64.tar.gz -C /usr/local \
          && git clone https://github.com/huaweicloud/cloudeye-exporter \
            && export PATH=$PATH:/usr/local/go/bin \
              && export GO111MODULE=on \
                && export GOPROXY=https://goproxy.cn,direct \
                  && export GONOSUMDB=* \
                    && cd cloudeye-exporter \
                      && go build
CMD ["/cloudeye-exporter/cloudeye-exporter -config=/tmp/clouds.yml"]
```

Step 2 Build an image. The image name is **cloudeye-exporter** and the image version is 1.0.

```
docker build --network host . -t cloudeye-exporter:1.0
```

Step 3 Push the image to SWR.

1. (Optional) Log in to the SWR console, choose **Organizations** in the navigation pane, and click **Create Organization** in the upper right corner of the page.

   Skip this step if you already have an organization.

2. In the navigation pane, choose **My Images** and then click **Upload Through Client**. On the page displayed, click **Generate a temporary login command** and click  to copy the command.

3. Run the login command copied in the previous step on the cluster node. If the login is successful, the message "Login Succeeded" is displayed.

4. Tag the **cloudeye-exporter** image.

   docker tag *{Image name 1:Tag 1}|{Image repository address}|{Organization name}|{Image name 2:Tag 2}*

   - *{Image name 1:Tag 1}*: name and tag of the local image to be uploaded.
   - *{Image repository address}*: The domain name at the end of the login command in **2** is the image repository address, which can be obtained on the SWR console.
   - *{Organization name}*: name of the organization created in **1**.
   - *{Image name 2:Tag 2}*: desired image name and tag to be displayed on the SWR console.

   The following is an example:

docker tag cloudeye-exporter:1.0 swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/cloudeye-exporter:1.0

5. Push the image to the image repository.

   docker push *{Image repository address}/{Organization name}/{Image name 2:Tag 2}*

   The following is an example:

   docker push swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/cloudeye-exporter:1.0

   The following information will be returned upon a successful push:

   ```
   ...
   030***: Pushed
   1.0: digest: sha256:eb7e3bbd*** size: **
   ```

   To view the pushed image, go to the SWR console and refresh the **My Images** page.

----End

Deploying the Exporter

Prometheus can dynamically monitor pods if you add Prometheus annotations to the pods (the default path is **/metrics**). This section uses **cloudeye-exporter** as an example.

Common annotations in Prometheus are as follows:

- **prometheus.io/scrape**: If the value is **true**, the pod will be monitored.
- **prometheus.io/path**: URL from which the data is collected. The default value is **/metrics**.
- **prometheus.io/port**: port number of the endpoint to collect data from.
- **prometheus.io/scheme**: Defaults to **http**. If HTTPS is configured for security purposes, change the value to **https**.

Step 1 Use kubectl to connect to the cluster.

Step 2 Create a secret, which will be used by **cloudeye-exporter** for authentication.

1. Create the **clouds.yml** file with the following content:

   ```
   global:
     prefix: "huaweicloud"
     scrape_batch_size: 10
     port: ":8087"
     metric_path: "/metrics"
   auth:
     auth_url: "https://iam.ap-southeast-1.myhuaweicloud.com/v3"
     project_name: "ap-southeast-1"
     access_key: "********"
     secret_key: "**********"
     region: "ap-southeast-1"
   ```

   Parameters in the preceding content are described as follows:

   - **auth_url**: indicates the IAM endpoint, which can be obtained from **Regions and Endpoints**.
   - **project_name**: indicates the project name. On the **My Credential** page, view the project name and project ID in the **Projects** area.
   - **access_key** and **secret_key**: You can obtain them from **Access Keys**.

- **region**: indicates the region name, which must correspond to the project in **project_name**.

2. Obtain the Base64-encrypted string of the preceding file.

```
cat clouds.yml | base64 -w0 ;echo
```

3. Create the **clouds-secret.yaml** file with the following content:

```
apiVersion: v1
kind: Secret
data:
  clouds.yml: ICAga*****  # Replace it with the Base64-encrypted string.
metadata:
  annotations:
    description: ''
  name: 'clouds.yml'
  namespace: default  # Namespace where the key is located.
  labels: {}
type: Opaque
```

4. Create a secret.

```
kubectl apply -f clouds-secret.yaml
```

Step 3 Create the **cloudeye-exporter-deployment.yaml** file with the following content:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: cloudeye-exporter
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cloudeye-exporter
      version: v1
  template:
    metadata:
      labels:
        app: cloudeye-exporter
        version: v1
    spec:
      volumes:
        - name: vol-166055064743016314
          secret:
            secretName: clouds.yml
            defaultMode: 420
      containers:
        - name: container-1
          image: swr.ap-southeast-1.myhuaweicloud.com/cloud-develop/cloudeye-exporter:1.0
          command:
            - /cloudeye-exporter/cloudeye-exporter
            - '-config=/tmp/clouds.yml'
          resources: {}
          volumeMounts:
            - name: vol-166055064743016314
              readOnly: true
              mountPath: /tmp
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: default-secret
      schedulerName: default-scheduler
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
```



```
    maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

Create the preceding workload.

```
kubectl apply -f cloudeye-exporter-deployment.yaml
```

Step 4 Create the **cloudeye-exporter-service.yaml** file.

```
apiVersion: v1
kind: Service
metadata:
  name: cloudeye-exporter
  namespace: default
  labels:
    app: cloudeye-exporter
    version: v1
  annotations:
    prometheus.io/port: '8087'
    prometheus.io/scrape: 'true'
    prometheus.io/path: "/metrics"
    prometheus.io/scheme: "http"
spec:
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 8087
      targetPort: 8087
  selector:
    app: cloudeye-exporter
    version: v1
  type: ClusterIP
```

Create the preceding Service.

```
kubectl apply -f cloudeye-exporter-service.yaml
```

----End

Interconnecting with Prometheus

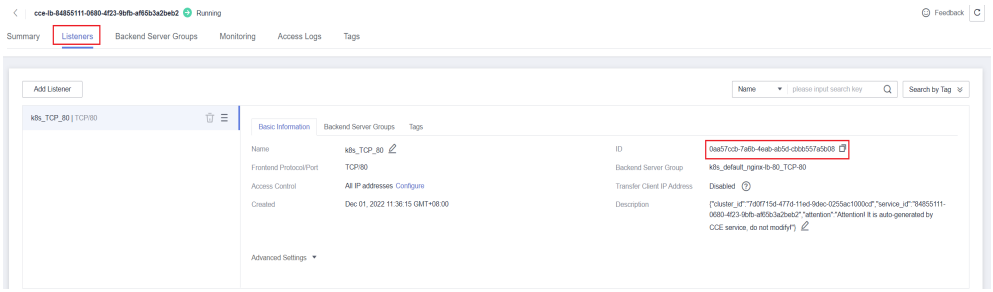
After collecting monitoring data, Prometheus needs to convert the data into the Kubernetes metric API for the HPA controller to perform auto scaling.

In this example, the ELB metrics associated with the workload need to be monitored. Therefore, the target workload must use the Service or ingress of the **LoadBalancer** type.

Step 1 View the access mode of the workload to be monitored and obtain the ELB listener ID.

1. On the CCE cluster console, choose **Networking**. On the **Services** or **Ingresses** tab page, view the Service or ingress of the **LoadBalancer** type and click the load balancer to access the load balancer page.

2. On the **Listeners** tab, view the listener corresponding to the workload and copy the listener ID.



Step 2 Use kubectl to connect to the cluster and add Prometheus configurations. In this example, collect load balancer metrics. For details about advanced usage, see **Configuration**.

1. Create the **prometheus-additional.yaml** file, add the following content to the file, and save the file:

```
- job_name: elb_metric
  params:
    services: ['SYS.ELB']
  kubernetes_sd_configs:
    - role: endpoints
  relabel_configs:
    - action: keep
      regex: '8087'
      source_labels:
        - __meta_kubernetes_service_annotation_prometheus_io_port
    - action: replace
      regex: ([^:]+)(?::\d+)?;(\d+)
      replacement: $1:$2
      source_labels:
        - __address__
        - __meta_kubernetes_service_annotation_prometheus_io_port
      target_label: __address__
    - action: labelmap
      regex: __meta_kubernetes_service_label_(.+)
    - action: replace
      source_labels:
        - __meta_kubernetes_namespace
      target_label: kubernetes_namespace
    - action: replace
      source_labels:
        - __meta_kubernetes_service_name
      target_label: kubernetes_service
```

2. Use the preceding configuration file to create a secret named **additional-scrape-configs**.

```
kubectl create secret generic additional-scrape-configs --from-file prometheus-additional.yaml -n monitoring --dry-run=client -o yaml | kubectl apply -f -
```

3. Modify the Prometheus node object.

```
kubectl edit prometheus server -n monitoring
```

Add the following content to the **spec** field and save the file:

```
spec:
  additionalScrapeConfigs:
    key: prometheus-additional.yaml
    name: additional-scrape-configs
```

4. Check whether the modification has taken effect.

```
kubectl get secret prometheus-server -n monitoring -o jsonpath="{.data['prometheus\.yaml\.gz']}" | base64 --decode | gzip -d | grep -A3 elb
```

If any command output is displayed, the modification has taken effect.

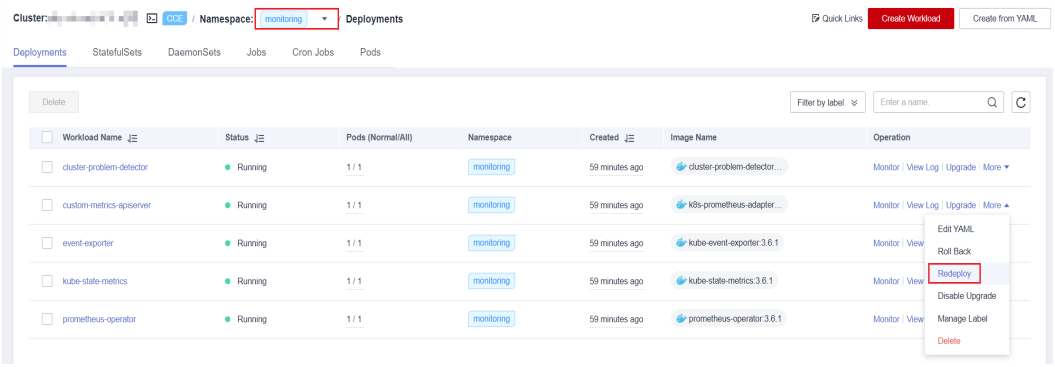
Step 3 Add the configmap configuration of **custom-metrics-apiserver** to **user-adapter-config**. (In earlier versions, the name of this configuration item is **adapter-config**.)

```
kubectl edit configmap user-adapter-config -nmonitoring
```

Add the following content under the **rules** field and save the file. Replace the listener ID obtained in **Step 1** with the value of **seriesQuery**.

```
apiVersion: v1
data:
  config.yaml: |-
    rules:
    - metricsQuery: sum(<<.Series>>{<<.LabelMatchers>>}) by (<<.GroupBy>>)
      resources:
        overrides:
          kubernetes_namespace:
            resource: namespace
          kubernetes_service:
            resource: service
      name:
        matches: huaweicloud_sys_elb_(.*)
        as: "elb01_${1}"
      seriesQuery: '{lbaas_listener_id="94424*****"}' # ELB listener ID
    ...
```

Step 4 Redeploy the **custom-metrics-apiserver** workload in the **monitoring** namespace.



----End

Creating an HPA Policy

After the data reported by the exporter to Prometheus is converted into the Kubernetes metric API by using the Prometheus adapter, you can create an HPA policy for auto scaling.

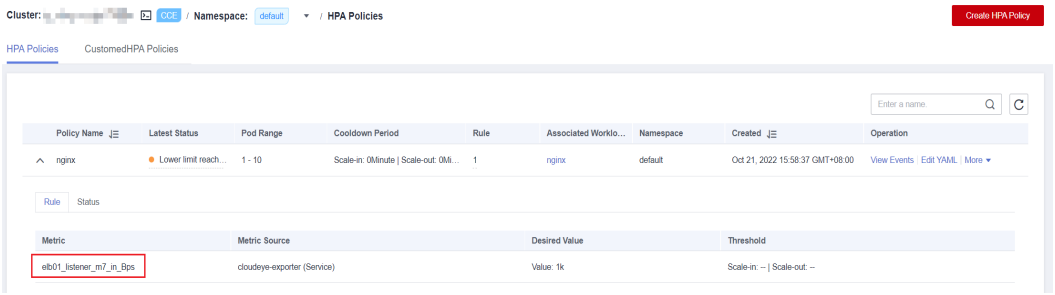
Step 1 Create an HPA policy. The inbound traffic of the ELB load balancer is used to trigger scale-out. When the value of **m7_in_Bps** (inbound traffic rate) exceeds 1000, the nginx Deployment will be scaled.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 1
  maxReplicas: 10
```



```
metrics:
  - type: Object
    object:
      metric:
        name: elb01_listener_m7_in_Bps
      describedObject:
        apiVersion: v1
        kind: Service
        name: cloudeye-exporter
      target:
        type: Value
        value: 1000
```

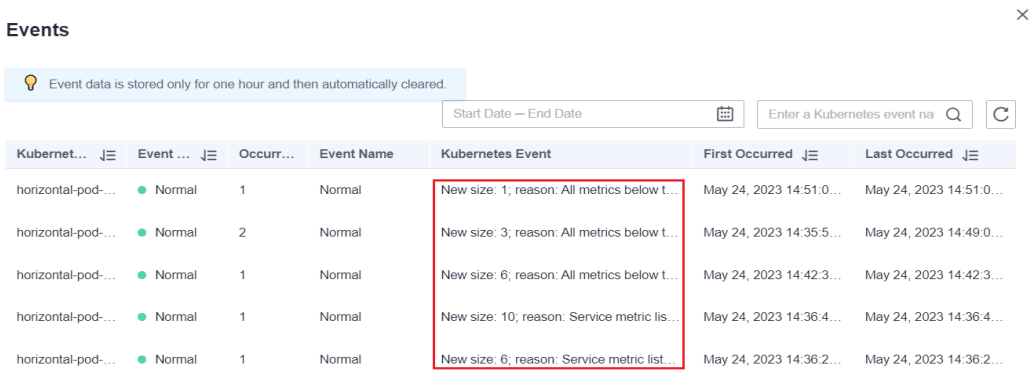
Figure 7-3 Created HPA Policy



Step 2 After the HPA policy is created, perform a pressure test on the workload (accessing the pods through ELB). Then, the HPA controller determines whether scaling is required based on the configured value.

In the **Events** dialog box, obtain scaling records in the **Kubernetes Event** column.

Figure 7-4 Scaling events



----End

ELB Listener Metrics

The following table lists the ELB listener metrics that can be collected using the method described in this section.

Table 7-3 ELB listener metrics

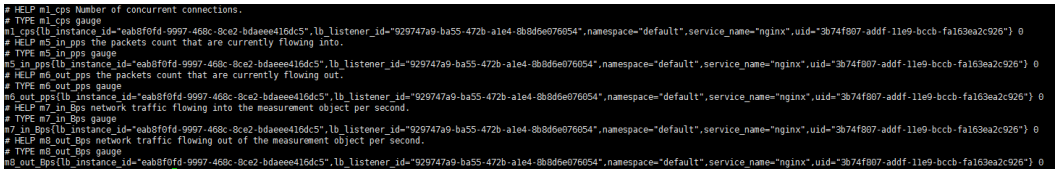
Metric	Name	Unit	Description
m1_cps	Concurrent Connections	Count	Number of concurrent connections processed by a load balancer.
m1e_server_rps	Reset Packets from Backend Servers	Count/Second	Number of reset packets sent from the backend server to clients. These reset packages are generated by the backend server and then forwarded by load balancers.
m1f_lvs_rps	Reset Packets from Load Balancers	Count/Second	Number of reset packets sent from load balancers.
m21_client_rps	Reset Packets from Clients	Count/Second	Number of reset packets sent from clients to the backend server. These reset packages are generated by the clients and then forwarded by load balancers.
m22_in_bandwidth	Inbound Bandwidth	bit/s	Inbound bandwidth of a load balancer.
m23_out_bandwidth	Outbound Bandwidth	bit/s	Outbound bandwidth of a load balancer.
m2_act_conn	Active Connections	Count	Number of current active connections.
m3_inact_conn	Inactive Connections	Count	Number of current inactive connections.
m4_ncps	New Connections	Count	Number of current new connections.
m5_in_pps	Incoming Packets	Count	Number of packets sent to a load balancer.
m6_out_pps	Outgoing Packets	Count	Number of packets sent from a load balancer.
m7_in_Bps	Inbound Rate	byte/s	Number of incoming bytes per second on a load balancer.
m8_out_Bps	Outbound Rate	byte/s	Number of outgoing bytes per second on a load balancer.

Appendix: Developing an Exporter

Prometheus periodically calls the **/metrics** API of the exporter to obtain metric data. Applications only need to report monitoring data through **/metrics**. You can

select a Prometheus client in a desired language and integrate it into applications to implement the **/metrics** API. For details about the client, see **Prometheus CLIENT LIBRARIES**. For details about how to write the exporter, see **WRITING EXPORTERS**.

The monitoring data must be in the format that Prometheus supports. Each data record provides the ELB ID, listener ID, namespace where the Service is located, Service name, and Service UID as labels, as shown in the following figure.



To obtain the preceding data, perform the following steps:

Step 1 Obtain all Services.

The **annotations** field in the returned information contains the ELB associated with the Service.

- kubernetes.io/elb.id
- kubernetes.io/elb.class

Step 2 Use APIs in **Querying Listeners** to get the listener ID based on the load balancer ID obtained in the previous step.

Step 3 Obtain the ELB monitoring data.

The ELB monitoring data is obtained using the CES APIs described in **Querying Monitoring Data of Multiple Metrics** . For details about ELB monitoring metrics, see **Monitoring Metrics**. Example:

- **m1_cps**: number of concurrent connections
- **m5_in_pps**: number of incoming data packets
- **m6_out_pps**: number of outgoing data packets
- **m7_in_Bps**: incoming rate
- **m8_out_Bps**: outgoing rate

Step 4 Aggregate data in the format that Prometheus supports and expose the data through the **/metrics** API.

The Prometheus client can easily call the **/metrics** API. For details, see **CLIENT LIBRARIES**. For details about how to develop an exporter, see **WRITING EXPORTERS**.

----End

8 Monitoring

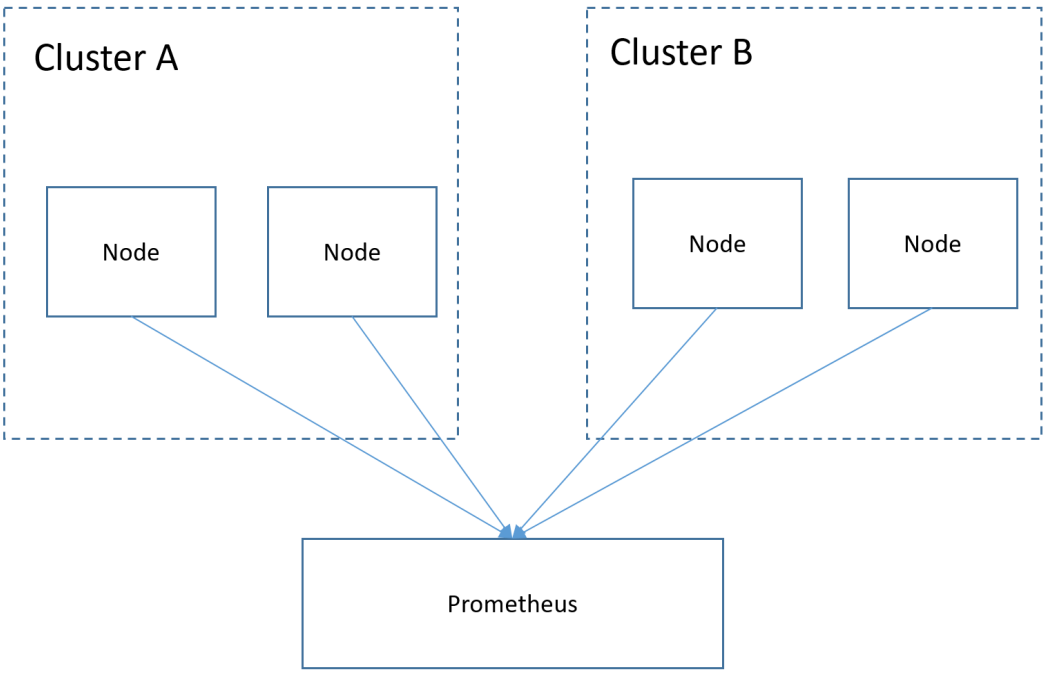
8.1 Using Prometheus for Multi-cluster Monitoring

Application Scenarios

Generally, a user has different clusters for different purposes, such as production, testing, and development. To monitor, collect, and view metrics of these clusters, you can deploy a set of Prometheus.

Solution Architecture

Multiple clusters are connected to the same Prometheus monitoring system, as shown in the following figure. This reduces maintenance and resource costs and facilitates monitoring information aggregation.



Prerequisites

- The target cluster has been created.
- Prometheus has been properly connected to the target cluster.
- Prometheus has been installed on a Linux host using a binary file. For details, see **Installation**.

Procedure

Step 1 Obtain the **bearer_token** information of the target cluster.

1. Create the RBAC permission in the target cluster.

   Log in to the background node of the target cluster and create the **prometheus_rbac.yaml** file.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus-test
  namespace: kube-system

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-test
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  - services
  - endpoints
  - pods
  - nodes/proxy
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - "extensions"
  resources:
    - ingresses
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - configmaps
  - nodes/metrics
  verbs:
  - get
- nonResourceURLs:
  - /metrics
  verbs:
  - get
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus-test
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
```



```
  name: prometheus-test
subjects:
- kind: ServiceAccount
  name: prometheus-test
  namespace: kube-system
```

Run the following command to create the RBAC permission:

kubectl apply -f prometheus_rbac.yaml

2. Obtain the **bearer_token** information of the target cluster.

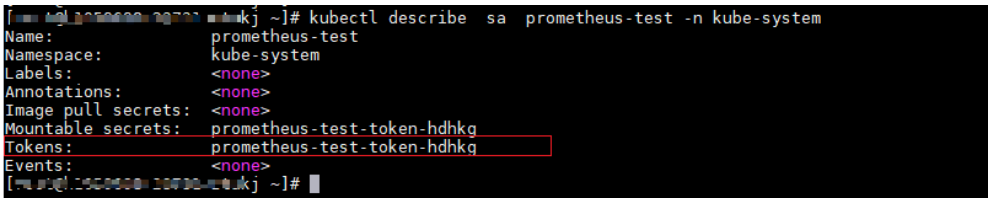
   NOTE

   - In clusters earlier than v1.21, a token is obtained by mounting the secret of the service account to a pod. Tokens obtained this way are permanent. This approach is no longer recommended starting from version 1.21. Service accounts will stop auto creating secrets in clusters from version 1.25.

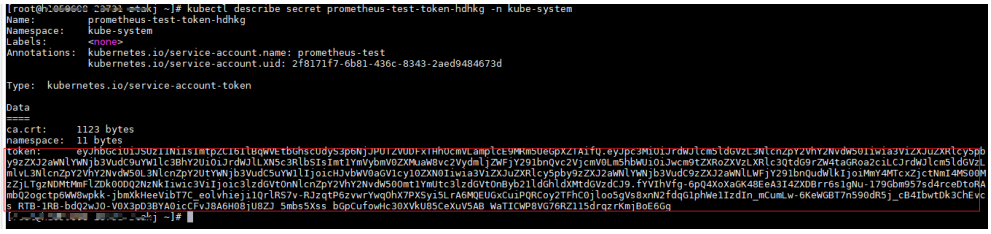
     In clusters of version 1.21 or later, you can use the **TokenRequest** API to **obtain the token** and use the projected volume to mount the token to the pod. Such tokens are valid for a fixed period. When the mounting pod is deleted, the token automatically becomes invalid. For details, see **Service Account Token Security Improvement**.
   - If you need a token that never expires, you can also **manually manage secrets for service accounts**. Although a permanent service account token can be manually created, you are advised to use a short-lived token by calling the **TokenRequest** API for higher security.

   Obtain the **serviceaccount** information.

   kubectl describe sa prometheus-test -n kube-system



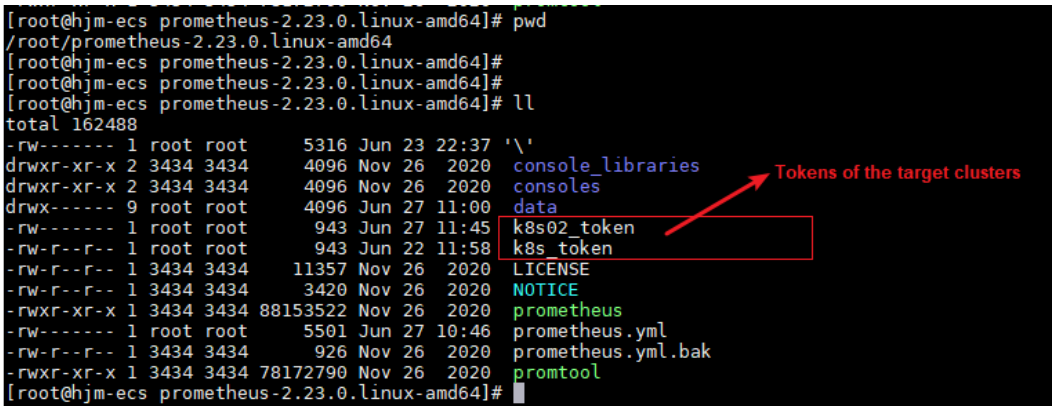
   kubectl describe secret prometheus-test-token-hdhkg -n kube-system



   Record the token value, which is the **bearer_token** information to be collected.

Step 2 Configure **bearer_token** information.

Log in to the host where Prometheus is located, go to the Prometheus installation directory, and save the token information of the target cluster in a file.



Step 3 Configure a Prometheus monitoring job.

The example job monitors container metrics. To monitor other metrics, you can add jobs and compile capture rules.

```
 - job_name: k8s_cAdvisor
   scheme: https
   bearer_token_file: k8s_token # Token file in the previous step.
   tls_config:
     insecure_skip_verify: true
   kubernetes_sd_configs:  # kubernetes automatic discovery configuration
   - role: node    # Automatic discovery of the node type
     bearer_token_file: k8s_token # Token file in the previous step
     api_server: https://192.168.0.153:5443  # API server address of the Kubernetes cluster
     tls_config:
       insecure_skip_verify: true   # Skip the authentication on the server.
   relabel_configs:  ## Modify the existing label of the target cluster before capturing metrics.
   - target_label: __address__
     replacement: 192.168.0.153:5443
     action: replace
     ## Convert metrics_path to /api/v1/nodes/${1}/proxy/metrics/cadvisor.
     # Obtain data from kubelet using the API server proxy.
   - source_labels: [__meta_kubernetes_node_name]   # Specifies the source label to be processed.
     regex: (.+)    # Matched value of the source label. (.+) indicates that any value of the source label can be matched.
     target_label: __metrics_path__     # Specifies the label to be replaced.
     replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor  # Indicates the new label, that is, the value of __metrics_path__. ${1} indicates the value that matches the regular expression, that is, node name.
   - target_label: cluster
     replacement: xxxxx   ## (Optional) Enter the cluster information.

### The following job monitors another cluster.
 - job_name: k8s02_cAdvisor
   scheme: https
   bearer_token_file: k8s02_token # Token file in the previous step
   tls_config:
     insecure_skip_verify: true
   kubernetes_sd_configs:
   - role: node
     bearer_token_file: k8s02_token # Token file in the previous step
     api_server: https://192.168.0.147:5443  # API server address of the Kubernetes cluster
     tls_config:
       insecure_skip_verify: true   # Skip the authentication on the server.
   relabel_configs:  ## Modify the existing label of the target cluster before capturing metrics.
   - target_label: __address__
     replacement: 192.168.0.147:5443
     action: replace

   - source_labels: [__meta_kubernetes_node_name]
     regex: (.+)
     target_label: __metrics_path__
     replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor
```



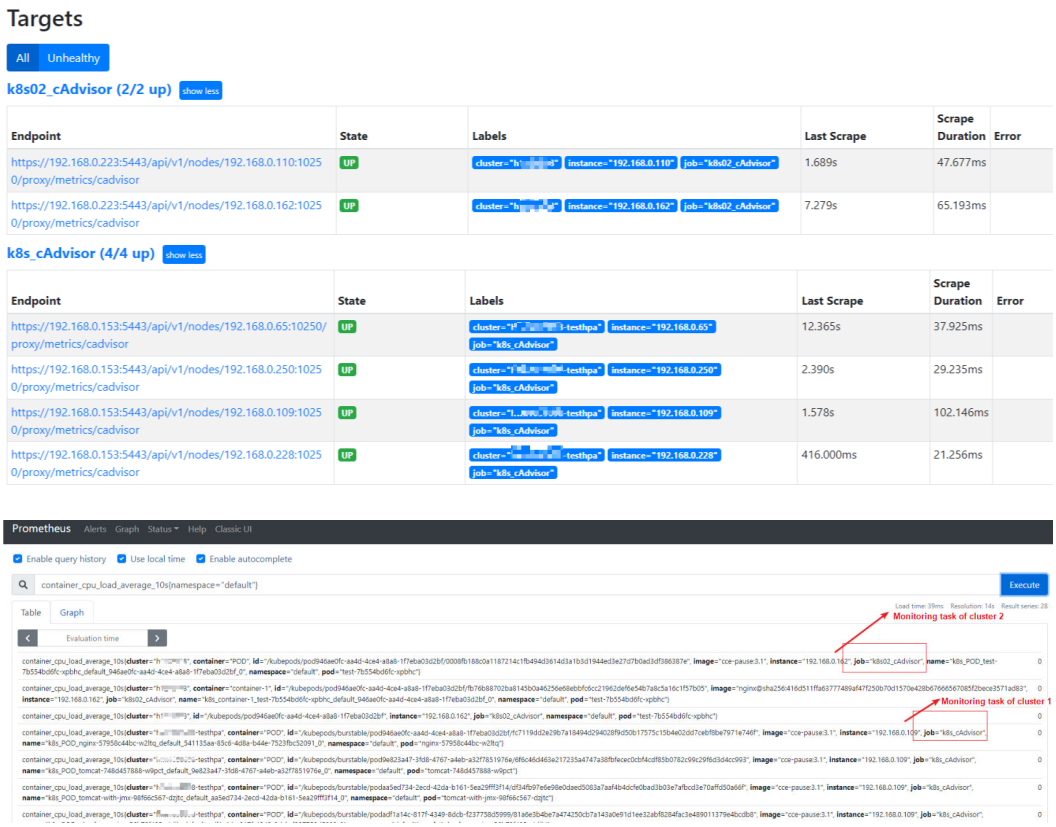
```
  - target_label: cluster
    replacement: xxxx    ## (Optional) Enter the cluster information.
```

Step 4 Enable Prometheus.

After the configuration, enable Prometheus.

./prometheus --config.file=prometheus.yml

Step 5 Log in to Prometheus and view the monitoring information.



----End

8.2 Using dcgm-exporter to Monitor GPU Metrics

Application Scenarios

If a cluster contains GPU nodes, learn about the GPU resources used by GPU applications, such as the GPU usage, memory usage, running temperature, and power. You can configure auto scaling policies or set alarm rules based on the obtained GPU metrics. This section walks you through how to observe GPU resource usage based on open source Prometheus and DCGM Exporter. For more details about DCGM Exporter, see **DCGM Exporter**.

Prerequisites

- You have created a cluster and there are GPU nodes and GPU related services running in the cluster.

- The CCE AI Suite (NVIDIA GPU) and Cloud Native Cluster Monitoring add-ons have been installed in the cluster.
  - CCE AI Suite (NVIDIA GPU) is a device management add-on that supports GPUs in containers. To use GPU nodes in the cluster, this add-on must be installed. Select and install the corresponding GPU driver based on the GPU type and CUDA version.
  - Cloud Native Cluster Monitoring monitors the cluster metrics. During the installation, you can interconnect this add-on with Grafana to gain a better observability of your cluster.

    NOTE

    - Set the deployment mode of Cloud Native Cluster Monitoring to the server mode.
    - The configuration for interconnecting with Grafana is supported by the Cloud Native Cluster Monitoring add-on of a version earlier than 3.9.0. For the add-on of version 3.9.0 or later, if Grafana is required, use the Grafana add-on separately..

Collecting GPU Monitoring Metrics

This section describes how to deploy the dcgm-exporter component in the cluster to collect GPU metrics and expose GPU metrics through port 9400.

Step 1 Log in to a node that has been bound with an EIP.

Step 2 Pull the dcgm-exporter image to the local host. The image address comes from the DCGM official example. For details, see **https://github.com/NVIDIA/dcgm-exporter/blob/main/dcgm-exporter.yaml**.

```
docker pull nvcr.io/nvidia/k8s/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04
```

Step 3 Push the dcgm-exporter image to SWR.

1. (Optional) Log in to the SWR console, choose **Organization Management** in the navigation pane, and click **Create Organization** in the upper right corner to create an organization.

   Skip this step if you already have an organization.
2. In the navigation pane, choose **My Images** and then click **Upload Through Client**. On the page displayed, click **Generate a temporary login command** and click  to copy the command.
3. Run the login command copied in the previous step on the cluster node. If the login is successful, the message "Login Succeeded" is displayed.
4. Add a tag to the dcgm-exporter image.

   docker tag *{Image name 1:Tag 1}|{Image repository address}|{Organization name}|{Image name 2:Tag 2}*

   - *{Image name 1:Tag 1}*: name and tag of the local image to be uploaded.
   - *{Image repository address}*: The domain name at the end of the login command in **2** is the image repository address, which can be obtained on the SWR console.
   - *{Organization name}*: name of the organization created in **1**.
   - *{Image name 2:Tag 2}*: desired image name and tag to be displayed on the SWR console.

The following is an example:

```
docker tag nvcr.io/nvidia/k8s/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04 swr.cn-east-3.myhuaweicloud.com/container/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04
```

5. Push the image to the image repository.

   docker push *{Image repository address}/{Organization name}/{Image name 2:Tag 2}*

   The following is an example:

   ```
   docker push swr.cn-east-3.myhuaweicloud.com/container/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04
   ```

   The following information will be returned upon a successful push:

   ```
   489a396b91d1: Pushed
   ...
   c3f11d77a5de: Pushed
   3.0.4-3.0.0-ubuntu20.04: digest: sha256:bd2b1a73025*** size: 2414
   ```

6. To view the pushed image, go to the SWR console and refresh the **My Images** page.

Step 4 Deploy dcgm-exporter.

When deploying dcgm-exporter on CCE, add some specific configurations to monitor GPU information. The detailed YAML file is as follows. The information in red is important.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: "dcgm-exporter"
  namespace: "monitoring"      # Select a namespace as required.
  labels:
    app.kubernetes.io/name: "dcgm-exporter"
    app.kubernetes.io/version: "3.0.0"
spec:
  updateStrategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app.kubernetes.io/name: "dcgm-exporter"
      app.kubernetes.io/version: "3.0.0"
  template:
    metadata:
      labels:
        app.kubernetes.io/name: "dcgm-exporter"
        app.kubernetes.io/version: "3.0.0"
      name: "dcgm-exporter"
    spec:
      containers:
      - image: "swr.cn-east-3.myhuaweicloud.com/container/dcgm-exporter:3.0.4-3.0.0-ubuntu20.04"   # The SWR image address of dcgm-exporter. The address is the image address in 5.
        env:
        - name: "DCGM_EXPORTER_LISTEN"                  # Service port number
          value: ":9400"
        - name: "DCGM_EXPORTER_KUBERNETES"              # Supports mapping of Kubernetes metrics to pods.
          value: "true"
        - name: "DCGM_EXPORTER_KUBERNETES_GPU_ID_TYPE"   # GPU ID type. The value can be uid or device-name.
          value: "device-name"
        name: "dcgm-exporter"
        ports:
        - name: "metrics"
          containerPort: 9400
        resources:      # Request and limit resources as required.
          limits:
            cpu: '200m'
            memory: '256Mi'
```



```
        requests:
          cpu: 100m
          memory: 128Mi
      securityContext:      # Enable the privilege mode for the dcgm-exporter container.
        privileged: true
        runAsNonRoot: false
        runAsUser: 0
      volumeMounts:
      - name: "pod-gpu-resources"
        readOnly: true
        mountPath: "/var/lib/kubelet/pod-resources"
      - name: "nvidia-install-dir-host"      # The environment variables configured in the dcgm-exporter
image depend on the file in the /usr/local/nvidia directory of the container.
        readOnly: true
        mountPath: "/usr/local/nvidia"
    volumes:
    - name: "pod-gpu-resources"
      hostPath:
        path: "/var/lib/kubelet/pod-resources"
    - name: "nvidia-install-dir-host"       # The directory where the GPU driver is installed.
      hostPath:
        path: "/opt/cloud/cce/nvidia"       #If the GPU add-on version is 2.0.0 or later, replace the driver
installation directory with /usr/local/nvidia.
    affinity:       # Label generated when CCE creates GPU nodes. You can set node affinity for this
component based on this label.
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: accelerator
              operator: Exists
---
kind: Service
apiVersion: v1
metadata:
  annotations:     # The following annotations enable Prometheus to automatically discover and extract
metrics data.
    prometheus.io/port: "9400"
    prometheus.io/scrape: "true"
  name: "dcgm-exporter"
  namespace: "monitoring"      # Select a namespace as required.
  labels:
    app.kubernetes.io/name: "dcgm-exporter"
    app.kubernetes.io/version: "3.0.0"
spec:
  selector:
    app.kubernetes.io/name: "dcgm-exporter"
    app.kubernetes.io/version: "3.0.0"
  ports:
  - name: "metrics"
    port: 9400
```

Step 5 Monitor application GPU metrics.

1. Run the following command to check whether the dcgm-exporter is running properly:

   ```
   kubectl get po -n monitoring -owide
   ```

   Information similar to the following is displayed:

   ```
   # kubectl get po -n monitoring -owide
   NAME                                  READY   STATUS    RESTARTS   AGE   IP            NODE
   NOMINATED NODE   READINESS GATES
   alertmanager-alertmanager-0           0/2     Pending   0          19m   <none>        <none>
   <none>           <none>
   custom-metrics-apiserver-5bb67f4b99-grxhq   1/1     Running   0          19m   172.16.0.6
   192.168.0.73   <none>           <none>
   dcgm-exporter-hkr77                   1/1     Running   0          17m   172.16.0.11    192.168.0.73
   <none>           <none>
   grafana-785cdcd47-9jlgr               1/1     Running   0          19m   172.16.0.9     192.168.0.73
   ```



```
<none>           <none>
kube-state-metrics-647b6585b8-6l2zm        1/1     Running   0          19m   172.16.0.8
192.168.0.73   <none>           <none>
node-exporter-xvk82                        1/1     Running   0          19m   192.168.0.73   192.168.0.73
<none>           <none>
prometheus-operator-5ff8744d5f-mhbqv       1/1     Running   0          19m   172.16.0.7
192.168.0.73   <none>           <none>
prometheus-server-0                        2/2     Running   0          19m   172.16.0.10    192.168.0.73
<none>           <none>
```

2. Call the **dcgm-exporter** API to verify the collected application GPU information.

   172.16.0.11 indicates the pod IP address of the dcgm-exporter.

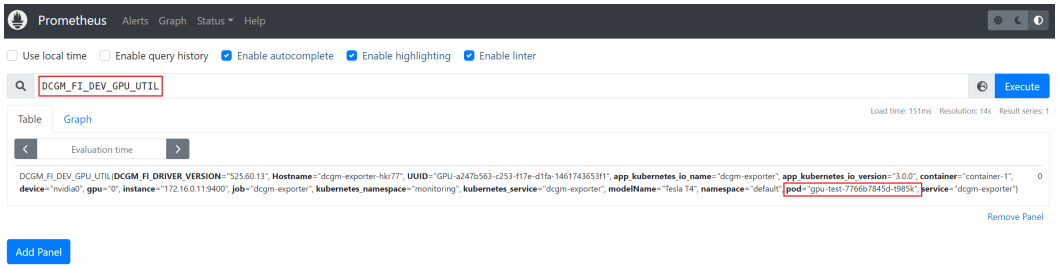
   ```
   curl 172.16.0.11:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
   ```

   ```
   [roo~ ~]# curl 172.16.0.11:9400/metrics | grep DCGM_FI_DEV_GPU_UTIL
     % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                    Dload  Upload   Total   Spent    Left  Speed
   100  5514    0  5514    0     0  5384k      # HELP DCGM_FI_DEV_GPU_UTIL GPU utilization (in %).
   0# TYPE DCGM_FI_DEV_GPU_UTIL gauge
    --DCGM_FI_DEV_GPU_UTIL{gpu="0",UUID="GPU-a247b563-c253-f17e-d1fa-1461743653f1",device="nvidia0",modelName="Tesla T4",Hostname="dcgm-exporter-hkr77",DCGM_FI_DRIVER_VE
   RSION="525.60.13",container="container-1",namespace="default",pod="gpu-test-7766b7845d-t985k"} 0
   :--:-- --:--:-- --:--:-- 5384k
   ```

Step 6 View metric monitoring information on the Prometheus page.

After prometheus and the related add-on are installed, a ClusterIP Service is created by default. To allow external systems to access the Service, create a NodePort or a LoadBalancer Service. For details, see **Monitoring Custom Metrics Using Prometheus**.

As shown in the following figure, you can view the GPU usage and other related metrics on the GPU node. For more GPU metrics, see **Observable Metrics**.



Step 7 Log in to the Grafana page to view GPU information.

If you have installed Grafana, you can import **NVIDIA DCGM Exporter dashboard** to display GPU metrics.

For details, see **Manage dashboards**.

----End

Observable Metrics

The following table lists some observable GPU metrics. For details about more metrics, see **Field Identifiers**.

Table 8-1 Usage

Metric Name	Metric Type	Unit	Description
DCGM_FI_DEV_GPU_UTIL	Gauge	%	GPU usage
DCGM_FI_DEV_MEM_COPY_UTIL	Gauge	%	Memory usage
DCGM_FI_DEV_ENC_UTIL	Gauge	%	Encoder usage
DCGM_FI_DEV_DEC_UTIL	Gauge	%	Decoder usage

Table 8-2 Memory

Metric Name	Metric Type	Unit	Description
DCGM_FI_DEV_FB_FREE	Gauge	MB	Number of remaining frame buffers. The frame buffer is called VRAM.

Metric Name	Metric Type	Unit	Description
DCGM_FI_DEV_FB_USED	Gauge	MB	Number of used frame buffers. The value is the same as the value of **memory-usage** in the **nvidia-smi** command.

Table 8-3 Temperature and power

Metric Name	Metric Type	Unit	Description
DCGM_FI_DEV_GPU_TEMP	Gauge	°C	Current GPU temperature of the device
DCGM_FI_DEV_POWER_USAGE	Gauge	W	Power usage of the device

9 Cluster

9.1 Configuring a CCE Cluster

When you use CCE to create a Kubernetes cluster, there are multiple configuration options and terms. This sections compares the key configurations for CCE clusters and provides recommendations to help you create a cluster that better suits your needs.

Cluster Types

CCE supports CCE Turbo clusters and CCE standard clusters to meet your requirements. This section describes the differences between these two types of clusters.

Table 9-1 Cluster types

Category	Subcategory	CCE Turbo Cluster	CCE Standard Cluster
Cluster	Positioning	Next-gen container cluster designed for Cloud Native 2.0, with accelerated computing, networking, and scheduling	Standard cluster for common commercial use
	Node type	Deployment of VMs and bare-metal servers	Deployment of VMs and bare-metal servers
Networking	Model	**Cloud Native Network 2.0**: applies to large-scale and high-performance scenarios. Max networking scale: 2,000 nodes	**Cloud Native Network 1.0**: applies to common, smaller-scale scenarios. • Tunnel network model • VPC network model

Category	Subcategory	CCE Turbo Cluster	CCE Standard Cluster
	Performance	Flattens the VPC network and container network into one, achieving zero performance loss.	Overlays the VPC network with the container network, causing certain performance loss.
	Container network isolation	Associates pods with security groups. Unifies security isolation in and out the cluster via security groups' network policies.	• Tunnel network model: supports network policies for intra-cluster communications. • VPC network model: supports no isolation.
Security	Isolation	• Physical machine: runs Kata containers, allowing VM-level isolation. • VM: runs common containers, isolated by cgroups.	Runs common containers, isolated by cgroups.

Cluster Versions

Due to the fast iteration, many bugs are fixed and new features are added in the new Kubernetes versions. The old versions will be gradually eliminated. When creating a cluster, select the latest commercial version supported by CCE.

Network Models

This section describes the network models supported by CCE clusters. You can select one model based on your requirements.

NOTICE

After clusters are created, the network models cannot be changed. Exercise caution when selecting the network models.

Table 9-2 Network model comparison

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Application scenarios	• Common container service scenarios • Scenarios that do not have high requirements on network latency and bandwidth	• Scenarios that have high requirements on network latency and bandwidth • Containers can communicate with VMs using a microservice registration framework, such as Dubbo and CSE.	• Scenarios that have high requirements on network latency, bandwidth, and performance • Containers can communicate with VMs using a microservice registration framework, such as Dubbo and CSE.
Core technology	OVS	IPvlan and VPC route	VPC ENI/sub-ENI
Applicable clusters	CCE standard cluster	CCE standard cluster	CCE Turbo cluster
Network isolation	Kubernetes native NetworkPolicy for pods	No	Pods support security group isolation.
Passthrough networking	No	No	Yes
IP address management	• The container CIDR block is allocated separately. • CIDR blocks are divided by node and can be dynamically allocated (CIDR blocks can be dynamically added after being allocated.)	• The container CIDR block is allocated separately. • CIDR blocks are divided by node and statically allocated (the CIDR block cannot be changed after a node is created).	The container CIDR block is divided from the VPC subnet and does not need to be allocated separately.

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Network performance	Performance loss due to VXLAN encapsulation	No tunnel encapsulation. Cross-node packets are forwarded through VPC routers, delivering performance equivalent to that of the host network.	The container network is integrated with the VPC network, eliminating performance loss.
Networking scale	A maximum of 2000 nodes are supported.	Suitable for small- and medium-scale networks due to the limitation on VPC routing tables. It is recommended that the number of nodes be less than or equal to 1000. Each time a node is added to the cluster, a route is added to the VPC routing tables (including the default and custom ones). Therefore, the cluster scale is limited by the VPC routing tables. For details about routing tables, see **Constraints**.	A maximum of 2000 nodes are supported.

For more information, see **Container Network Models Overview**.

Cluster CIDR Blocks

There are node CIDR blocks, container CIDR blocks, and Service CIDR blocks in CCE clusters. When planning network addresses, note that:

- These three types of CIDR blocks cannot overlap with each other. Otherwise, a conflict will occur. All subnets (including those created from the secondary CIDR block) in the VPC where the cluster resides cannot conflict with the container and Service CIDR blocks.
- There are sufficient IP addresses in each CIDR block.
  - The IP addresses in a node CIDR block must match the cluster scale. Otherwise, nodes cannot be created due to insufficient IP addresses.

  - The IP addresses in a container CIDR block must match the service scale. Otherwise, pods cannot be created due to insufficient IP addresses.

In complex scenarios, for example, multiple clusters use the same VPC or clusters are interconnected across VPCs, determine the number of VPCs, the number of subnets, the container CIDR blocks, and the communication modes of the Service CIDR blocks. For details, see **Planning CIDR Blocks for a Cluster**.

Service Forwarding Modes

kube-proxy is a key component of a Kubernetes cluster. It is responsible for load balancing and forwarding between a Service and its backend pod.

CCE supports the iptables and IPVS forwarding modes.

- IPVS allows higher throughput and faster forwarding. It applies to scenarios where the cluster scale is large or the number of Services is large.
- iptables is the traditional kube-proxy mode. This mode applies to the scenario where the number of Services is small or there are a large number of short concurrent connections on the client.

If high stability is required and the number of Services is less than 2000, the iptables forwarding mode is recommended. In other scenarios, the IPVS forwarding mode is recommended.

For details, see **Comparing iptables and IPVS**.

Node Specifications

The minimum specifications of a node are 2 vCPUs and 4 GiB memory. Evaluate based on service requirements before configuring the nodes. However, using many low-specification ECSs is not the optimal choice. The reasons are as follows:

- The upper limit of network resources is low, which may result in a single-point bottleneck.
- Resources may be wasted. If each container running on a low-specification node needs a lot of resources, the node cannot run multiple containers and there may be idle resources in it.

Advantages of using large-specification nodes are as follows:

- The upper limit of the network bandwidth is high. This ensures higher resource utilization for high-bandwidth applications.
- Multiple containers can run on the same node, and the network latency between containers is low.
- The efficiency of pulling images is higher. This is because an image can be used by multiple containers on a node after being pulled once. Low-specifications ECSs cannot respond promptly because the images are pulled many times and it takes more time to scale these nodes.

Additionally, select a proper vCPU/memory ratio based on your requirements. For example, if a service container with large memory but fewer CPUs is used, configure the specifications with the vCPU/memory ratio of 1:4 for the node where the container resides to reduce resource waste.

Container Engines

CCE supports the containerd and Docker container engines. **containerd is recommended for its shorter traces, fewer components, higher stability, and less consumption of node resources**. Since Kubernetes 1.24, Dockershim is removed and Docker is no longer supported by default. For details, see **Kubernetes is Moving on From Dockershim: Commitments and Next Steps**. CCE clusters 1.27 do not support the Docker container engine.

Use containerd in typical scenarios. The Docker container engine is supported only in the following scenarios:

- Docker in Docker (usually in CI scenarios)
- Running the Docker commands on the nodes
- Calling Docker APIs

Node OS

Service container runtimes share the kernel and underlying calls of nodes. To ensure compatibility, select a Linux distribution version that is the same as or close to that of the final service container image for the node OS.

9.2 Creating an IPv4/IPv6 Dual-Stack Cluster in CCE

This section describes how to set up a VPC with IPv6 CIDR block and create a cluster and nodes with an IPv6 address in the VPC, so that the nodes can access the Internet.

Overview

IPv6 addresses are used to deal with the problem of IPv4 address exhaustion. If a worker node (such as an ECS) in the current cluster uses IPv4, the node can run in dual-stack mode after IPv6 is enabled. Specifically, the node has both IPv4 and IPv6 addresses, which can be used to access the intranet or public network.

Application Scenarios

- If your application needs to provide Services for users who use IPv6 clients, you can use IPv6 EIPs or the IPv4 and IPv6 dual-stack function.
- If your application needs to both provide Services for users who use IPv6 clients and analyze the access request data, you can use only the IPv4 and IPv6 dual-stack function.
- If internal communication is required between your application systems or between your application system and another system (such as the database system), you can use only the IPv4 and IPv6 dual-stack function.

For details about the dual stack, see **IPv4 and IPv6 Dual-Stack Network**.

Constraints

- Clusters that support IPv4/IPv6 dual stack:

Cluster Type	Cluster Network Model	Version	Remarks
CCE cluster	Container tunnel network	v1.15 or later	IPv4/IPv6 dual stack will be generally available for clusters of v1.23. ELB dual stack is not supported.
CCE Turbo cluster	Cloud Native 2.0 Network	v1.23.8-r0 or later v1.25.3-r0 or later	Currently, Kata containers do not support IPv4/IPv6 dual stack. Only ECS-VM or ECS-physical server (c6.22xlarge.4.physical or c7.32xlarge.4.physical) supports IPv4/IPv6 dual stack.

- Worker nodes and master nodes in Kubernetes clusters use IPv4 addresses to communicate with each other.
- If the Service type is set to **DNAT**, only IPv4 addresses are supported.
- Only one IPv6 address can be bound to each NIC.
- When IPv4/IPv6 dual stack is enabled for the cluster, DHCP unlimited lease cannot be enabled for the selected node subnet.
- If a dual-stack cluster is used, do not change the load balancer protocol version on the ELB console.

Step 1: Create a VPC

Before creating your VPCs, determine how many VPCs, the number of subnets, and what IP address ranges you will need. For details, see **Network Planning**.

NOTE

- The basic operations for IPv4 and IPv6 dual-stack networks are the same as those for IPv4 networks. Only some parameters are different.
- For details about the IPv6 billing policy, supported ECS types, and supported regions, see **IPv4 and IPv6 Dual-Stack Network**.

Perform the following operations to create a VPC named **vpc-ipv6** and its default subnet named **subnet-ipv6**.

1. Log in to the management console.
2. Click  in the upper left corner of the management console and select a region and a project.
3. Under **Networking**, select **Virtual Private Cloud**.
4. Click **Create VPC**.
5. Set the VPC and subnet parameters.

   When configuring a subnet, select **Enable** for **IPv6 CIDR Block** to automatically allocate an IPv6 CIDR block to the subnet. IPv6 cannot be disabled after the subnet is created. Currently, you are not allowed to specify a custom IPv6 CIDR block.

Table 9-3 VPC configuration parameters

Parameter	Description	Example Value
Region	Specifies the desired region. Regions are geographic areas that are physically isolated from each other. The networks inside different regions are not connected to each other, so resources cannot be shared across different regions. For lower network latency and faster access to your resources, select the region nearest you.	AP-Singapore
Name	VPC name.	vpc-ipv6
IPv4 CIDR Block	Specifies the Classless Inter-Domain Routing (CIDR) block of the VPC. The CIDR block of a subnet can be the same as the CIDR block for the VPC (for a single subnet in the VPC) or a subset (for multiple subnets in the VPC). The following CIDR blocks are supported: 10.0.0.0/8–24 172.16.0.0/12–24 192.168.0.0/16–24	192.168.0.0/16
Enterprise Project	When creating a VPC, you can add the VPC to an enabled enterprise project. An enterprise project facilitates project-level management and grouping of cloud resources and users. The name of the default project is **default**. For details about how to create and manage enterprise projects, see ***Enterprise Management User Guide***.	default
Tag (Advanced Settings)	Specifies the VPC tag, which consists of a key and value pair. You can add a maximum of ten tags for each VPC. The tag key and value must meet the requirements listed in **Table 9-5**.	• **Tag key**: vpc_key1 • **Key value**: vpc-01

Table 9-4 Subnet parameter description

Parameter	Description	Example Value
AZ	An AZ is a geographic location with independent power supply and network facilities in a region. AZs are physically isolated, and AZs in the same VPC are interconnected through an internal network.	AZ2
Name	Specifies the subnet name.	subnet-ipv6
IPv4 CIDR Block	Specifies the IPv4 CIDR block for the subnet. This value must be within the VPC CIDR range.	192.168.0.0/24
IPv6 CIDR Block	Select **Enable** for **IPv6 CIDR Block**. An IPv6 CIDR block will be automatically assigned to the subnet. IPv6 cannot be disabled after the subnet is created. Currently, you are not allowed to specify a custom IPv6 CIDR block.	N/A
Associated Route Table	Specifies the default route table to which the subnet will be associated. You can change the route table to a custom route table.	Default
Advanced Settings		
Gateway	Specifies the gateway address of the subnet. This IP address is used to communicate with other subnets.	192.168.0.1
DNS Server Address	By default, two DNS server addresses are configured. You can change them if necessary. When multiple IP addresses are available, separate them with a comma (,).	100.125.x.x
DHCP Lease Time	Specifies the period during which a client can use an IP address automatically assigned by the DHCP server. After the lease time expires, a new IP address will be assigned to the client. If a DHCP lease time is changed, the new lease automatically takes effect when half of the current lease time has passed. To make the change take effect immediately, restart the ECS or log in to the ECS to cause the DHCP lease to automatically renew. **CAUTION** When IPv4/IPv6 dual stack is enabled for the cluster, DHCP unlimited lease cannot be enabled for the selected node subnet.	365 days or 300 hours

Parameter	Description	Example Value
Tag	Specifies the subnet tag, which consists of a key and value pair. You can add a maximum of ten tags to each subnet. The tag key and value must meet the requirements listed in **Table 9-6**.	• **Tag key**: subnet_key1 • **Key value**: subnet-01

Table 9-5 VPC tag key and value requirements

Parameter	Requirement	Example Value
Tag key	• Cannot be left blank. • Must be unique in a VPC. • Can contain a maximum of 36 characters. • Can contain letters, digits, underscores (_), and hyphens (-).	vpc_key1
Tag value	• Can contain a maximum of 43 characters. • Can contain letters, digits, underscores (_), periods (.), and hyphens (-).	vpc-01

Table 9-6 Subnet tag key and value requirements

Parameter	Requirement	Example Value
Tag key	• Cannot be left blank. • Must be unique for each subnet. • Can contain a maximum of 36 characters. • Can contain letters, digits, underscores (_), and hyphens (-).	subnet_key1
Tag value	• Can contain a maximum of 43 characters. • Can contain letters, digits, underscores (_), periods (.), and hyphens (-).	subnet-01

6. Click **Create Now**.

Step 2: Create a CCE Cluster

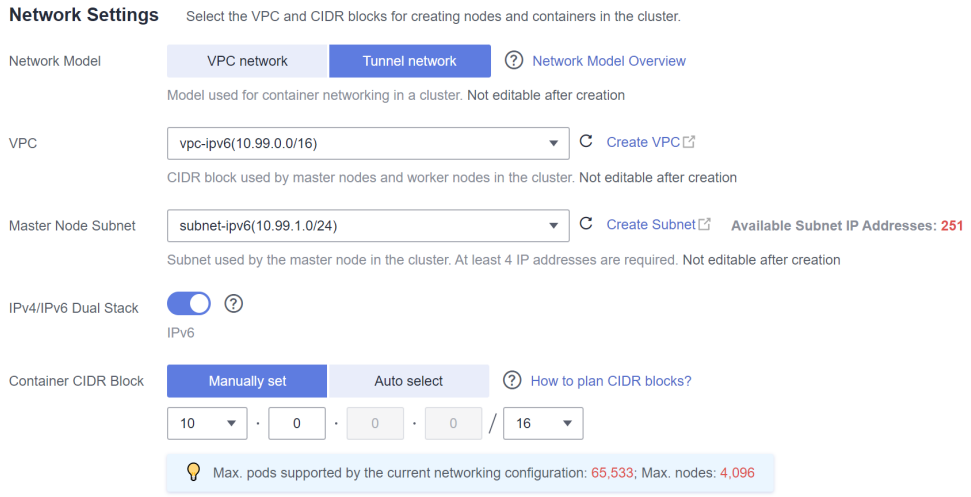
Creating a CCE cluster

1. Log in to the CCE console and create a cluster.

   Complete the network settings as follows. For other configurations, see **Buying a CCE Cluster**.

   - **Network Model**: Select **Tunnel network**.
   - **VPC**: Select the created VPC **vpc-ipv6**.
   - **Master Node Subnet**: Select a subnet with IPv6 enabled.
   - **IPv4/IPv6 Dual Stack**: Enable this function. After this function is enabled, cluster resources, including nodes and workloads, can be accessed through IPv6 CIDR blocks.
   - **Container CIDR Block**: A proper mask must be set for the container CIDR block. The mask determines the number of available nodes in the cluster. If the mask of the container CIDR block in the cluster is set improperly, there will be only a small number of available nodes in the cluster.

   Figure 9-1 Configuring network settings

2. Create a node.

   The CCE console displays the nodes that support IPv6. You can directly select a node. For details, see **Creating a Node**.

   After the creation is complete, access the cluster details page. Then, click the node name to go to the ECS details page and view the automatically allocated IPv6 address.

Step 3: Buy a Shared Bandwidth and Adding an IPv6 Address to It

By default, the IPv6 address can only be used for private network communication. If you want to use this IPv6 address to access the Internet or be accessed by IPv6 clients on the Internet, buy a shared bandwidth and add the IPv6 address to it.

If you already have a shared bandwidth, you can add the IPv6 address to the shared bandwidth without purchasing one.

Buying a Shared Bandwidth

1. Log in to the management console.

2. Click  in the upper left corner of the management console and select a region and a project.
3. Choose **Service List** > **Networking** > **Virtual Private Cloud**.
4. In the navigation pane, choose **Elastic IP and Bandwidth** > **Shared Bandwidths**.
5. In the upper right corner, click **Buy Shared Bandwidth**. On the displayed page, configure parameters following instructions.

Table 9-7 Parameters

Parameter	Description	Example Value
Billing Mode	Specifies the billing mode of a shared bandwidth. The billing mode can be: • **Yearly/Monthly**: You pay for the bandwidth by year or month before using it. No charges will be incurred for the bandwidth during its validity period. • **Pay-per-use**: You pay for the bandwidth based on the amount of time you use the bandwidth.	Yearly/ Monthly
Region	Specifies the desired region. Regions are geographic areas that are physically isolated from each other. The networks inside different regions are not connected to each other, so resources cannot be shared across different regions. For lower network latency and faster access to your resources, select the region nearest you.	AP-Singapore -
Billed By	Specifies the shared bandwidth billing factor.	Select **Bandwidth**.
Bandwidth	Specifies the shared bandwidth size in Mbit/s. The minimum bandwidth that can be purchased is 5 Mbit/s.	10
Name	Specifies the name of the shared bandwidth.	Bandwidth-001
Enterprise Project	When assigning the shared bandwidth, you can add the shared bandwidth to an enabled enterprise project. An enterprise project facilitates project-level management and grouping of cloud resources and users. The name of the default project is **default**. For details about how to create and manage enterprise projects, see ***Enterprise Management User Guide***.	default

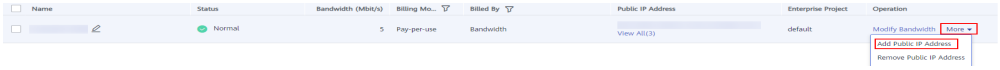
Parameter	Description	Example Value
Required Duration	Specifies the required duration of the shared bandwidth to be purchased. Configure this parameter only in yearly/monthly billing mode.	2 months

6. Click Next.

Adding an IPv6 Address to a Shared Bandwidth

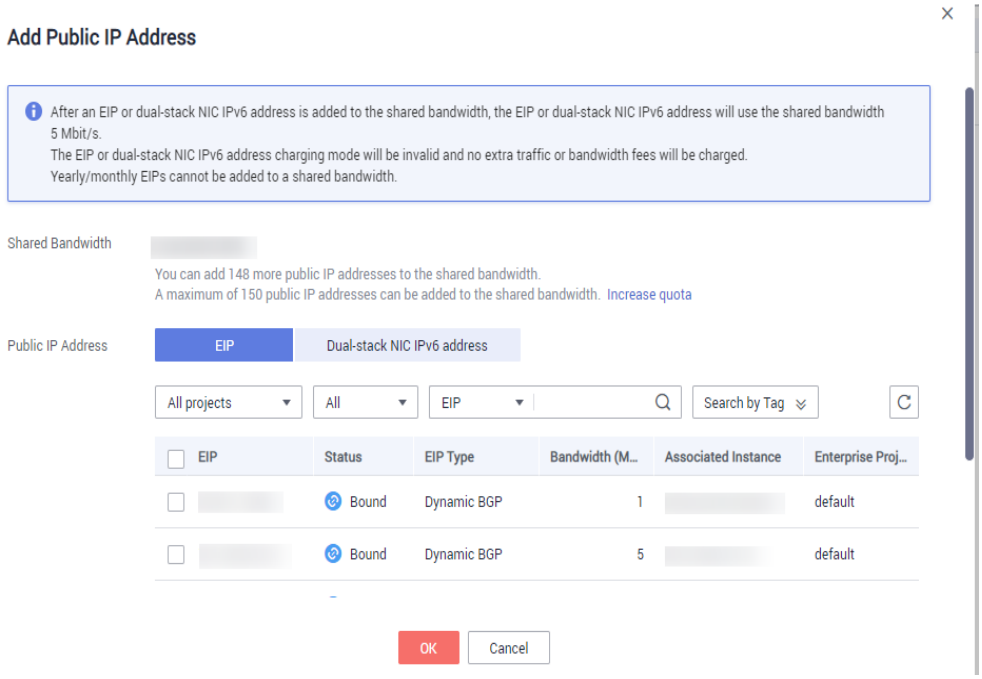
1. On the shared bandwidth list page, locate the row containing the target shared bandwidth and click **Add Public IP Address** in the **Operation** column.

   Figure 9-2 Adding an IPv6 address to a shared bandwidth

2. Add the IPv6 address to the shared bandwidth.

   Figure 9-3 Adding a dual-stack NIC IPv6 address

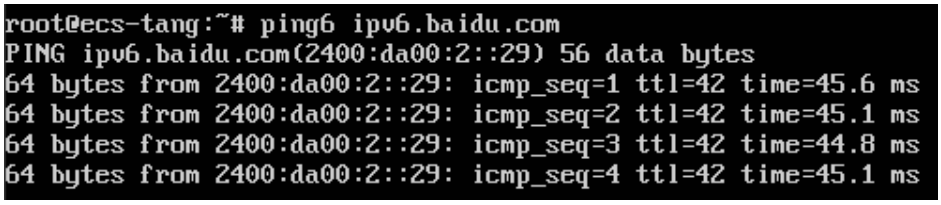
   

3. Click **OK**.

Verifying the Result

Log in to an ECS and ping an IPv6 address on the Internet to verify the connectivity. **ping6 ipv6.baidu.com** is used as an example here. The execution result is displayed in **Figure 9-4**.

Figure 9-4 Result verification



9.3 Creating a Custom CCE Node Image

Custom CCE node images are created using the open source tool **HashiCorp Packer** of v1.7.2 or later and the **open source plug-in**. The cce-image-builder template is provided to help you quickly build images.

Packer is used to create custom container images. It offers builders, provisioners, and post-processors that can be flexibly combined to automatically create image files concurrently through JSON or HCL template files.

Packer has the following advantages:

1. Automatic build process: You can use Packer configuration files to specify and automate the build process.
2. High compatibility with cloud platforms: Packer can interconnect with most cloud platforms and various third-party plug-ins.
3. Easy-to-use configuration files: Packer configuration files are simple and intuitive to write and read. Parameter definitions are easy to understand.
4. Diverse image build functions: Common functional modules are supported. For example, the provisioner supports the shell module in remote script execution, the file module in remote file transfer, and the breakpoint module for process pauses.

Constraints

- Suggestions on using CCE node images:
  - You are advised to use the default node images maintained by CCE. These images have passed strict tests and updated in a timely manner, providing better compatibility, stability, and security.
  - Use the base images provided by CCE to create custom images.
  - The component package on which nodes depend for running is preset in the base image. The package version varies with the cluster version. For custom images, CCE does not push component package updates.
- When customizing an image, exercise caution when modifying kernel parameters. Any improper kernel parameter modification will deteriorate the system running efficiency. For details about the reference values, see **Modifying Node Kernel Parameters**.

  Modifying the following kernel parameters will affect the system performance: **tcp_keepalive_time**, **tcp_max_tw_buckets**, **somaxconn, max_user_instances**, **max_user_watches**, **netdev_max_backlog**, **net.core.wmem_max**, and **net.core.rmem_max**.

  To modify node kernel parameters, fully verify the modification in a test environment before applying the modification to the production environment.

Precautions

- Before you create an image, prepare:
  - An ECS executor: An ECS x86 server is used as the Linux executor. You are advised to select CentOS7 and bind an EIP to it so that it can access the public network and install Packer.
  - Authentication credentials: Obtain the AK/SK of the tenant or user with required permissions. For details, see **How Do I Obtain an Access Key (AK/SK)**.
  - Security group: Packer creates a temporary ECS and uses a key pair to log in to the ECS using SSH. Ensure that **TCP:22** is enabled in the security group. For details, see **Security Group Configuration Examples**.
- When you create a custom node image, make sure:
  - You follow the instructions in this section to prevent unexpected problems.
  - You have the **sudo root** or **root** permissions required to log in to VMs created from base images.
- When the creation is complete:
  - The image creation process uses certain charging resources, including ECSs, EVS disks, EIPs, bandwidth, and IMS images. These resources are automatically released when the image is successfully created or fails to be created. Release the resources in time to ensure no charges are incurred unexpectedly.

Obtaining an Image ID

Step 1 Log in to the CCE console and click the cluster name to access the cluster console. In the navigation pane, choose **Nodes**.

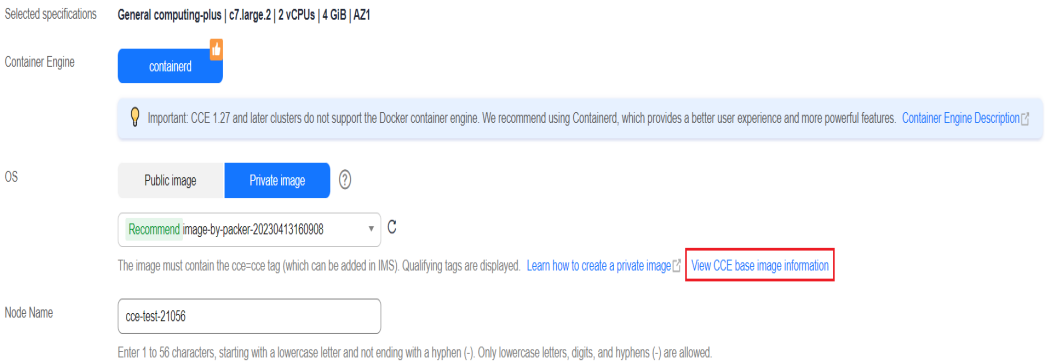
Step 2 Click **Create Node** in the upper right corner and select **OS** to **Private image**.

Step 3 Click **View CCE base image information**. In the displayed dialog box, copy the image ID.

NOTICE

The image ID varies depending on the region. If the region is changed, obtain the image ID again.

Figure 9-5 Obtaining an Image ID



----End

Creating a Node Image

Step 1 Download cce-image-builder.

Log in to the ECS executor, download and decompress cce-image-builder.

```
wget https://cce-north-4.obs.cn-north-4.myhuaweicloud.com/cce-image-builder/cce-image-builder.tgz

tar zvxf cce-image-builder.tgz
cd cce-image-builder/
```

NOTE

The cce-image-builder contains:

- turbo-node.pkr.hcl # Packer configuration template used for creating the image. For details about how to modify the template, see **Step 3**.
- scripts/* # CCE image creation preset in the template. Do not modify it. Otherwise, the image might become unavailable.
- user-scripts/* **# Custom package script directory preset in the template.** Take example.sh as an example. When you create a custom image, the image is automatically uploaded to the temporary server and executed.
- user-packages/* **# Custom package directory preset in the template.** Take **example.package** as an example. When you create a custom image, the image is automatically uploaded to **/tmp/example.package** in the temporary server.

Step 2 Install Packer.

Download and install the **HashiCorp Packer**. For details, see **Install Packer**.

NOTE

The Packer version needs to be 1.10.0.

Take the CentOS 7 executor as an example. Run the following command to automatically install Packer (**This example is for reference only. For detailed operations, see the official guide**):

```
# Configure the yum repository and install Packer.
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
sudo yum -y install packer-1.10.0

# Configure an alias to avoid duplicate Packer binary in the OS and check the Packer version.
rpm -q packer
alias packer=$(rpm -ql packer)
packer -v
```


Step 3 Define Packer template parameters.

The **cce-image-builder/turbo-node.pkr.hcl** file defines the process of building an image using Packer. For details, see **Packer Documentation**.

NOTE

- Parameters of **variables** or **variable**

  turbo-node.pkr.hcl defines the parameters required in the process of building an image. You can configure the parameters based on the live environment. For details, see **Table 1**.

- Parameter of **packer**

  required_plugins defines the add-on dependency of Packer, including the add-on source and version range. When you run **packer init**, the add-on is automatically downloaded and initialized.

```
packer {
  required_plugins {
    huaweicloud = {
      version = ">= 1.0.4"
      source  = "github.com/huaweicloud/huaweicloud"
    }
  }
}
```

- Parameter of **source**

  The preceding defined variables are referred to automatically configure the parameters required for creating an ECS.

- Parameter of **build**

  The scripts are executed from top to bottom. Common modules such as the file upload module and script execution shell module are supported. The corresponding scripts and files are stored in the **user-scripts** and **user-packages** directories, respectively, in **cce-image-builder**.

  Example:

```
build {
  sources = ["source.huaweicloud-ecs.builder"]

# Example:
  provisioner "file" {
    source      = "<source file path>"
    destination = "<destination file path>"
  }

  provisioner "shell" {
    scripts = [
      "<source script file: step1.sh>",
      "<source script file: step2.sh>"
    ]
  }

  provisioner "shell" {
    inline = ["echo foo"]
  }
}
```

Step 4 Configure environment variables.

Configure the following environment variables on the executor:

```
export REGION_NAME=xxx
export IAM_ACCESS_KEY=xxx
export IAM_SECRET_KEY=xxx
export ECS_VPC_ID=xxx
export ECS_NETWORK_ID=xxx
export ECS_SECGRP_ID=xxx
export CCE_SOURCE_IMAGE_ID=xxx
export PKR_VAR_ecs_flavor=xxx
```


Table 9-8 Variables configuration

Parameter	Description	Remarks
REGION_NAME	Region to which the project belongs	To obtain the region information, go to **My Credentials**.
IAM_ACCESS_KEY	Access key for user authentication	Apply for a temporary AK and delete it when the image is built successfully.
IAM_SECRET_KEY	Secret key for user authentication	Apply for a temporary SK and delete it when the image is built successfully.
ECS_VPC_ID	VPC ID	Used by the temporary ECS server, which must be the same as that of the executor
ECS_NETWORK_ID	Network ID of the subnet	Used by the temporary ECS server. It is recommended that the value be the same as that of the executor. It is not the subnet ID.
ECS_SECGRP_ID	Security group ID	Used by the temporary ECS. The public IP address of the executor must be allowed to pass through port 22 in the inbound direction of the security group to ensure that the executor can log in to the temporary ECS using SSH.
CCE_SOURCE_IMAGE_ID	Latest CCE node image ID	For details, see **Obtaining an Image ID**.
PKR_VAR_ecs_flavor	Specifications of a temporary ECS	Enter a node flavor supported by CCE. The recommended flavor is 2 vCPUs and 4 GiB memory or higher. For details about the flavor name, see **A Summary List of x86 ECS Specifications**.

Note: Retain the default values of other parameters. To change the values, refer to the description in the variable definition in **turbo-node.pkr.hcl** and configure the value using environment variables.

Use the ECS flavor variable **ecs_az** as an example. If no AZ is specified, select a random AZ. If you want to specify an AZ, configure an environment variable. The same applies to other parameters.

```
# export PKR_VAR_<variable name>=<variable value>
export PKR_VAR_ecs_az=xxx
```

Step 5 Customize scripts and files.

Compile scripts and files by referring to the file and shell modules defined by the **build** field in the **pkr.hcl** file, and store the scripts and files in the **user-scripts** and **user-packages** directories in **cce-image-builder**.

NOTICE

When customizing an image, exercise caution when modifying kernel parameters. Any improper kernel parameter modification will deteriorate the system running efficiency. For details about the reference values, see **Modifying Node Kernel Parameters**.

Modifying the following kernel parameters will affect the system performance: **tcp_keepalive_time**, **tcp_max_tw_buckets**, **somaxconn, max_user_instances**, **max_user_watches**, **netdev_max_backlog**, **net.core.wmem_max**, and **net.core.rmem_max**.

To modify node kernel parameters, fully verify the modification in a test environment before applying the modification to the production environment.

Step 6 Create a custom image.

After custom parameter settings, create an image. The creation will take 3 to 5 minutes.

```
make image
```

NOTE

In the encapsulation script **packer.sh**:

- Automatic access of hashicorp.com by Packer is disabled by default for privacy protection and security purposes.
  ```
  export CHECKPOINT_DISABLE=false
  ```
- The debugging detailed logs option is enabled by default for better visibility and traceability. The local Packer build logs **packer_{timestamp}.log** is specified so that the logs can be packed to the **/var/log/** directory during build. If sensitive information is involved, remove the related logic.
  ```
  export PACKER_LOG=1
  export PACKER_BUILD_TIMESTAMP=$(date +%Y%m%d%H%M%S)
  export PACKER_LOG_PATH="packer_$PACKER_BUILD_TIMESTAMP.log"
  ```

For details about Packer configuration, see **Configuring Packer**.

After the image is created, information similar to the following will display.

```
==> huaweicloud-ecs.builder: Setting a 15m0s timeout for the next provisioner...
==> huaweicloud-ecs.builder: Provisioning with shell script: /tmp/packer-shell1759174699
==> huaweicloud-ecs.builder: Setting a 15m0s timeout for the next provisioner...
==> huaweicloud-ecs.builder: Uploading packer_20210530185050.log => /var/log/packer_20210530185050.log
    huaweicloud-ecs.builder: packer_20210530185050.log 43.63 KiB / 43.50 KiB [==============================] 100.29% 0s
==> huaweicloud-ecs.builder: Stopping server: 9c901ac9-37b5-40af-934e-7190e6fa088e ...
    huaweicloud-ecs.builder: Waiting for server to stop: 9c901ac9-37b5-40af-934e-7190e6fa088e ...
==> huaweicloud-ecs.builder: Creating the image: image-by-packer-20210530185050
==> huaweicloud-ecs.builder: Waiting for image image-by-packer-20210530185050 to become available ...
    huaweicloud-ecs.builder: Image: 64e940f4-d674-4ae1-89cc-299501581c59
==> huaweicloud-ecs.builder: Deleted temporary floating IP '494617cc-a7c9-442a-b3e8-3b90c2c3f804' (94.74.101.22)
==> huaweicloud-ecs.builder: Terminating the source server: 9c901ac9-37b5-40af-934e-7190e6fa088e ...
==> huaweicloud-ecs.builder: Deleting volume: bf769e29-e1fd-407b-bbec-79f353a3e671 ...
==> huaweicloud-ecs.builder: Deleting temporary keypair: packer_60b36e0b-1f16-acc5-df04-d045aba70856 ...
Build 'huaweicloud-ecs.builder' finished after 3 minutes 53 seconds.

==> Wait completed after 3 minutes 53 seconds

==> Builds finished. The artifacts of successful builds are:
--> huaweicloud-ecs.builder: An image was created: 64e940f4-d674-4ae1-89cc-299501581c59
[Sun May 30 18:54:45 CST 2021] packer.sh finish.
```

Step 7 Clean up build files.

Clear the build files on the executor, mainly the authentication credentials in **turbo-node.pkr.hcl**.

- If the authentication credentials are temporary, directly release the executor.
- If they are built automatically, add post-processor in the configuration file to execute related operations.

----End

9.4 Executing the Post-installation Command During Node Creation

Background

When creating a node, use the post-installation commands to install tools or perform security hardening on the node. This section provides guidance for you to correctly use the post-installation scripts. To use advanced scripts, store the scripts in the OBS buckets to prevent the number of characters in the scripts from exceeding the upper limit. For details, see **Creating a Node Injection Script**.

Precautions

- Do not use the post-installation script that takes a long time to execute.

  The time limit to create a node in the CCE clusters is 30 minutes. If the node is not available within 30 minutes, it will be reclaimed. Therefore, do not run the post-installation script that takes a long time.
- Do not directly use the **reboot** command in the script.

  CCE executes the post-installation commands after installing mandatory components on the node. The node will be available only after the post-installation commands are executed. If you run **reboot** directly, the node may be restarted before its status is reported. As a result, it cannot reach the running state within 30 minutes, and a rollback due to timeout will be triggered. Therefore, do not run the **reboot** command.

  If you need to restart the node, perform the following operations:

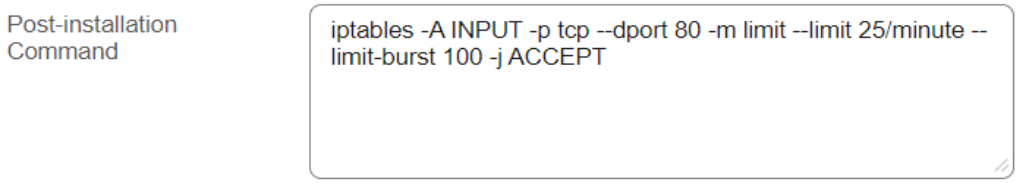
  - Run the **shutdown -r** *<time >* command in the script to delay the restart. For example, you can run **shutdown -r 1** to delay the restart for 1 minute.
  - After the node is available, manually restart it.

Procedure

Step 1 Log in to the CCE console. In the navigation pane, choose **Clusters**. Click the target cluster name to access the cluster console.

Step 2 Choose **Nodes** in the navigation pane, click the **Nodes** tab, click **Create Node** in the right corner, and configure the parameters.

Step 3 In the **Advanced Settings** area, enter the post-installation command.



For example, you can create iptables rules by running a post-installation command to allow a maximum of 25 TCP data packets to be addressed to port 80 per minute and allow a maximum of 100 data packets to be addressed to the port when the limit is exceeded to prevent DDoS attacks.

```
iptables -A INPUT -p tcp --dport 80 -m limit --limit 25/minute --limit-burst 100 -j ACCEPT
```

NOTE

The command example here is for reference only.

Step 4 After the configuration, enter the number of nodes to be purchased and click **Next: Confirm**.

Step 5 Click **Submit**.

----End

9.5 Creating a Node Injection Script

Background

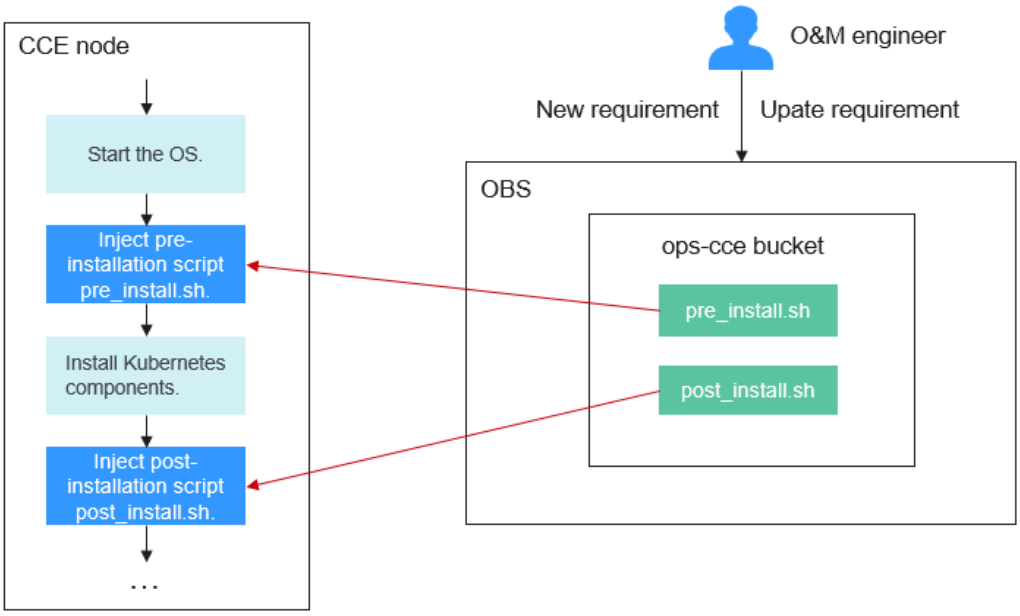
If you need to install some tools or perform custom security hardening on a node in advance, you need to inject some scripts when creating the node. CCE allows you to inject pre-installation and post-installation scripts when creating a node. However, this function has the following restrictions:

- The script characters are limited.
- The injected script content needs be frequently modified to meet various requirements and scenarios. However, CCE node pools have fixed scripts and do not support frequent modifications.

Solution

This section describes how to use CCE and OBS to provide a simplified, scalable, and easy-to-maintain solution so that you can operate CCE nodes as you want.

Store the scripts in the OBS buckets. During node pool creation, CCE automatically pulls and executes the stored scripts based on the configured injection scripts. In this way, the configuration of the CCE node pool does not need to be changed. If there are new requirements, you only need to update the scripts in the OBS buckets.



Suggestions on Maintaining OBS Buckets

- If there is no OBS bucket dedicated for O&M, create an OBS bucket dedicated for O&M.
- Create a multi-level directory **tools/cce** in the bucket to represent CCE-specific tools for easy maintenance. You can also store other tool scripts in this directory later.

Precautions

- If the custom operation implemented by the script fails, the normal service running is affected. You are advised to add a check program at the end of the script. If the check fails, stop the kubelet process in the post-installation script to prevent services from being scheduled to the node.

```
systemctl stop kubelet-monit
systemctl stop kubelet
```

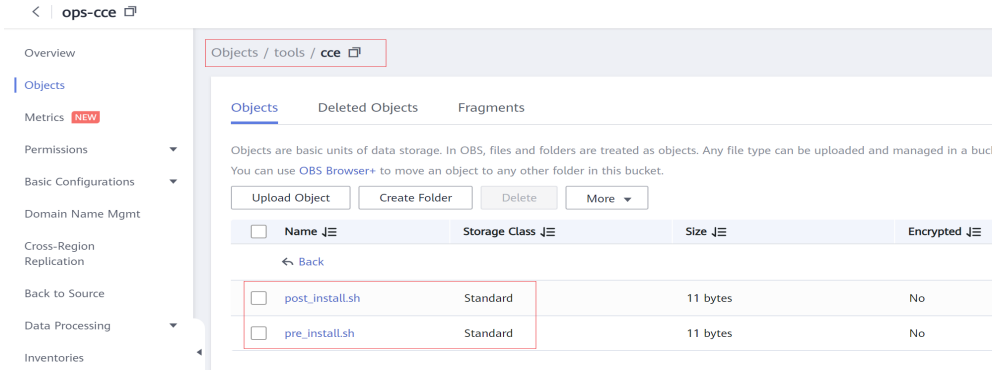
- Do not include sensitive information in the scripts to prevent information leakage.

Procedure

Step 1 Create an OBS bucket.

Step 2 Upload the pre-installation and post-installation scripts. The **pre_install.sh** and **post_install.sh** scripts are used as examples.

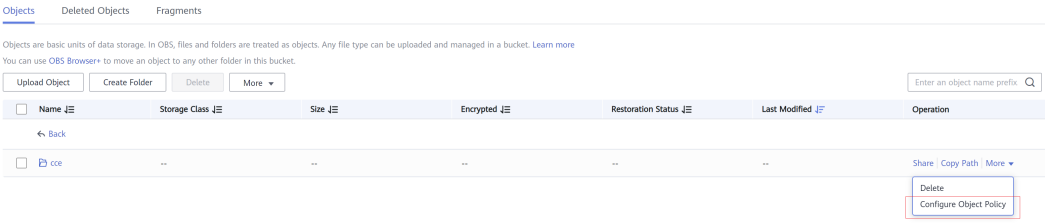
Figure 9-6 Uploading scripts



Step 3 Configure a read-only security policy for the scripts to ensure that the scripts can be downloaded without entering a password on the CCE nodes but cannot be downloaded from the Internet.

1. Configure a policy for the **tools/cce** directory and select a read-only policy template.

   Figure 9-7 Configuring an object policy



2. Modify the configuration information about the anonymous user, specified objects, and conditions.

   Figure 9-8 Anonymous user

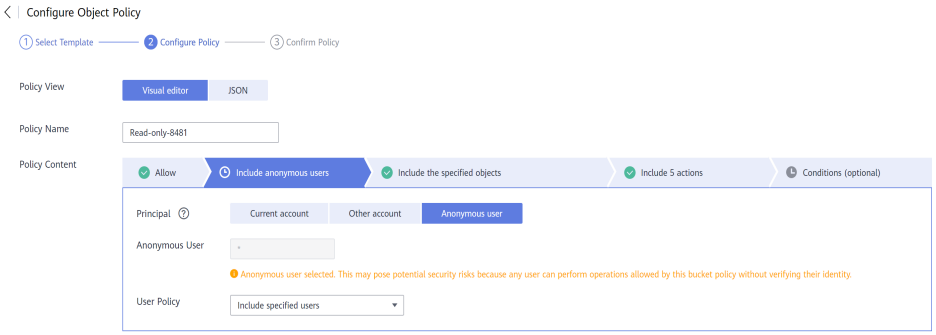


Figure 9-9 Specified objects

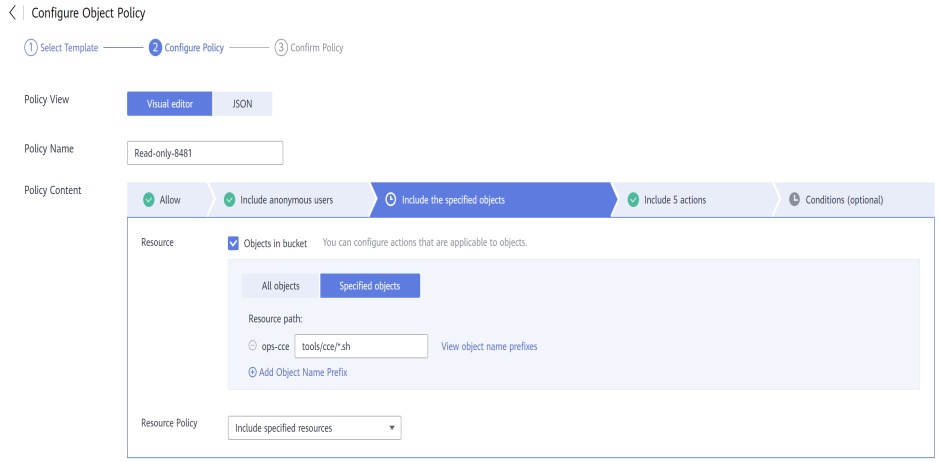
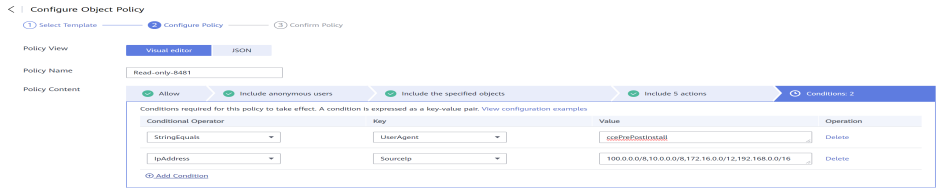


Figure 9-10 Conditions



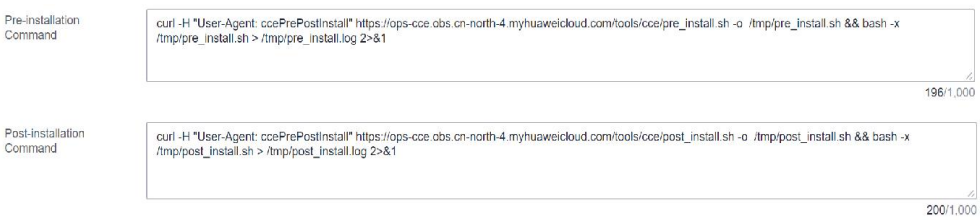
Two conditions are set: **UserAgent** and **SourceIp**.

- **UserAgent** functions in the similar way as a key. The specified User-Agent request header and the corresponding key value must be carried during access.
- **SourceIp** is used to prevent access from external networks. Set this parameter to **100.0.0.0/8,10.0.0.0/8,172.16.0.0/12,192.168.0.0/16,214.0.0.0/8**, among which **100.0.0.0/8** and **214.0.0.0/8** are private IP ranges, among whom **10.0.0.0/8**, **172.16.0.0/12**, and **192.168.0.0/16** are commonly used.

For details about how to configure an OBS policy, see **Configuring an Object Policy** and **Bucket Policy Parameters**.

Step 4 Configure the pre-installation script and post-installation script when creating a node pool on CCE.

Enter the following scripts in the **Advanced ECS Settings** on the **Create Node Pool** page:



In the scripts below, the **curl** command is run to download **pre_install.sh** and **post_install.sh** from OBS to the **/tmp** directory, and then **pre_install.sh** and **post_install.sh** are executed.

Pre-installation script:

```
curl -H "User-Agent: ccePrePostInstall" https://ops-cce.obs.ap-southeast-1.myhuaweicloud.com/tools/cce/pre_install.sh -o  /tmp/pre_install.sh && bash -x /tmp/pre_install.sh > /tmp/pre_install.log 2>&1
```

Post-installation script:

```
curl -H "User-Agent: ccePrePostInstall" https://ops-cce.obs.ap-southeast-1.myhuaweicloud.com/tools/cce/post_install.sh -o  /tmp/post_install.sh && bash -x /tmp/post_install.sh > /tmp/post_install.log 2>&1
```

NOTE

- The actual value of **User-Agent** must be configured based on the OBS bucket policy.
- The bucket address in the link must be configured based on site requirements.

----End

9.6 Connecting to Multiple Clusters Using kubectl

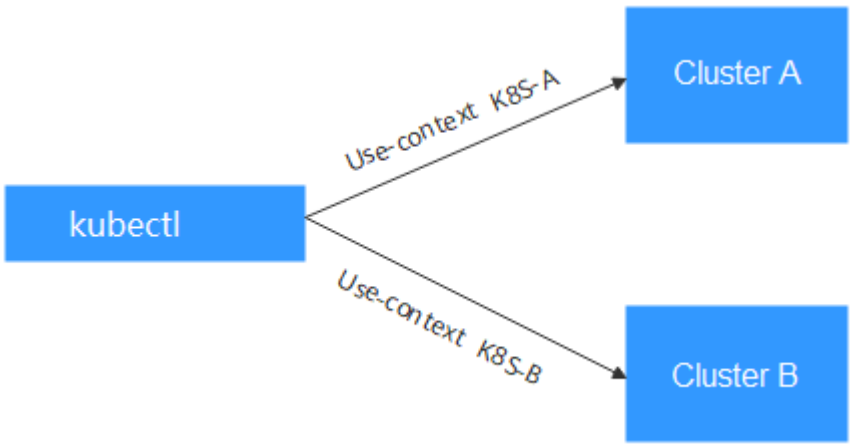
Background

When you have multiple CCE clusters, you may find it difficult to efficiently connect to all of them.

Solution

This section describes how to configure access to multiple clusters by modifying **kubeconfig.json**. The file describes multiple clusters, users, and contexts. To access different clusters, run the **kubectl config use-context** command to switch between contexts.

Figure 9-11 Using kubectl to connect to multiple clusters

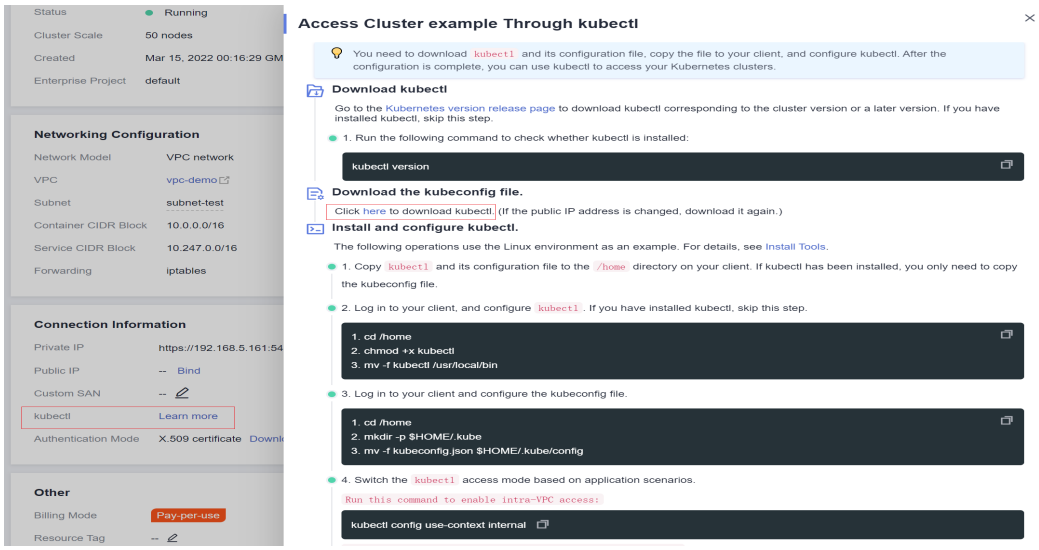


Prerequisites

kubectl can access multiple clusters.

Introduction to kubeconfig.json

kubeconfig.json is the configuration file of kubectl. You can download it on the cluster details page.



The content of kubeconfig.json is as follows:

```
{
   "kind": "Config",
   "apiVersion": "v1",
   "preferences": {},
   "clusters": [{
      "name": "internalCluster",
      "cluster": {
         "server": "https://192.168.0.85:5443",
         "certificate-authority-data": "LS0tLS1CRUULIE..."
      }
   }, {
      "name": "externalCluster",
      "cluster": {
         "server": "https://xxx.xxx.xxx.xxx:5443",
         "insecure-skip-tls-verify": true
      }
   }],
   "users": [{
      "name": "user",
      "user": {
         "client-certificate-data": "LS0tLS1CRUdJTiBDRVJ...",
         "client-key-data": "LS0tLS1CRUdJTiBS..."
      }
   }],
   "contexts": [{
      "name": "internal",
      "context": {
         "cluster": "internalCluster",
         "user": "user"
      }
   }, {
      "name": "external",
      "context": {
         "cluster": "externalCluster",
         "user": "user"
      }
   }],
   "current-context": "external"
}
```

It mainly consists of three sections.

- **clusters**: describes the cluster information, mainly the access address of the cluster.

- **users**: describes information about the users who access the cluster. It includes the **client-certificate-data** and **client-key-data** certificate files.
- **contexts**: describes the configuration contexts. You switch between contexts to access different clusters. A context is associated with **user** and **cluster**, that is, it defines which user accesses which cluster.

The preceding kubeconfig.json defines the private network address and public network address of the cluster as two clusters with two different contexts. You can switch the context to use different addresses to access the cluster.

Configuring Access to Multiple Clusters

The following steps walk you through the procedure of configuring access to two clusters by modifying kubeconfig.json.

This example configures only the public network access to the clusters. If you want to access multiple clusters over private networks, retain the **clusters** field and ensure that the clusters can be accessed over private networks. Its configuration is similar to that described in this example.

Step 1 Download kubeconfig.json of the two clusters and delete the lines related to private network access, as shown in the following figure.

- Cluster A:

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "externalCluster",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "external",
        "context": {
            "cluster": "externalCluster",
            "user": "user"
        }
    }],
    "current-context": "external"
}
```

- Cluster B:

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "externalCluster",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
```



```
  "users": [{
     "name": "user",
     "user": {
        "client-certificate-data": "LS0tLS1CRUdJTxM...",
        "client-key-data": "LS0rTUideUdJTiB...."
     }
  }],
  "contexts": [{
     "name": "external",
     "context": {
        "cluster": "externalCluster",
        "user": "user"
     }
  }],
  "current-context": "external"
}
```

The preceding files have the same structure except that the **client-certificate-data** and **client-key-data** fields of **user** and the **clusters.cluster.server** field are different.

Step 2 Modify the **name** field as follows:

- Cluster A:

```
{
  "kind": "Config",
  "apiVersion": "v1",
  "preferences": {},
  "clusters": [ {
     "name": "Cluster-A",
     "cluster": {
        "server": "https://119.xxx.xxx.xxx:5443",
        "insecure-skip-tls-verify": true
     }
  }],
  "users": [{
     "name": "Cluster-A-user",
     "user": {
        "client-certificate-data": "LS0tLS1CRUdJTxM...",
        "client-key-data": "LS0tLS1CRUdJTiB...."
     }
  }],
  "contexts": [{
     "name": "Cluster-A-Context",
     "context": {
        "cluster": "Cluster-A",
        "user": "Cluster-A-user"
     }
  }],
  "current-context": "Cluster-A-Context"
}
```

- Cluster B:

```
{
  "kind": "Config",
  "apiVersion": "v1",
  "preferences": {},
  "clusters": [ {
     "name": "Cluster-B",
     "cluster": {
        "server": "https://124.xxx.xxx.xxx:5443",
        "insecure-skip-tls-verify": true
     }
  }],
  "users": [{
     "name": "Cluster-B-user",
     "user": {
        "client-certificate-data": "LS0tLS1CRUdJTxM...",
        "client-key-data": "LS0rTUideUdJTiB...."
     }
```



```
    }],
    "contexts": [{
        "name": "Cluster-B-Context",
        "context": {
            "cluster": "Cluster-B",
            "user": "Cluster-B-user"
        }
    }],
    "current-context": "Cluster-B-Context"
}
```

Step 3 Combine these two files.

The file structure remains unchanged. Combine the contents of **clusters**, **users**, and **contexts** as follows:

```
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [ {
        "name": "Cluster-A",
        "cluster": {
            "server": "https://119.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    },
     {
        "name": "Cluster-B",
        "cluster": {
            "server": "https://124.xxx.xxx.xxx:5443",
            "insecure-skip-tls-verify": true
        }
    }],
    "users": [{
        "name": "Cluster-A-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0tLS1CRUdJTiB...."
        }
    },
    {
        "name": "Cluster-B-user",
        "user": {
            "client-certificate-data": "LS0tLS1CRUdJTxM...",
            "client-key-data": "LS0rTUideUdJTiB...."
        }
    }],
    "contexts": [{
        "name": "Cluster-A-Context",
        "context": {
            "cluster": "Cluster-A",
            "user": "Cluster-A-user"
        }
    },
    {
        "name": "Cluster-B-Context",
        "context": {
            "cluster": "Cluster-B",
            "user": "Cluster-B-user"
        }
    }],
    "current-context": "Cluster-A-Context"
}
```

----End

Verification

Run the following command to copy the conbined file to the kubectl configuration path:

mkdir -p $HOME/.kube

mv -f kubeconfig.json $HOME/.kube/config

Run the kubectl commands to check whether the two clusters can be connected.

```
# kubectl config use-context Cluster-A-Context
Switched to context "Cluster-A-Context".
# kubectl cluster-info
Kubernetes control plane is running at https://119.xxx.xxx.xxx:5443
CoreDNS is running at https://119.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

# kubectl config use-context Cluster-B-Context
Switched to context "Cluster-B-Context".
# kubectl cluster-info
Kubernetes control plane is running at https://124.xxx.xxx.xxx:5443
CoreDNS is running at https://124.xxx.xxx.xxx:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

9.7 Selecting a Data Disk for the Node

When a node is created, a data disk is attached by default for a container runtime and kubelet. For details, see **Data Disk Space Allocation**. The data disk used by the container runtime and kubelet cannot be detached, and the default capacity is 100 GiB. To cut costs, you can adjust the disk capacity to the minimum of 20 GiB or reduce the disk capacity attached to a node to the minimum of 10 GiB.

NOTICE

Adjusting the size of the data disk used by the container runtime and kubelet may incur risks. You are advised to evaluate the capacity adjustment and then perform the operations described in this section.

- If the disk capacity is too small, the image pull may fail. If different images need to be frequently pulled on the node, you are not advised to reduce the data disk capacity.
- Before a cluster upgrade, the system checks whether the data disk usage exceeds 95%. If the usage is high, the cluster upgrade may be affected.
- If Device Mapper is used, the disk capacity may be insufficient. You are advised to use the OverlayFS or select a large-capacity data disk.
- For dumping logs, application logs must be stored in a separate disk to prevent insufficient storage capacity of the dockersys volume from affecting service running.
- After reducing the data disk capacity, you are advised to install the npd add-on in the cluster to detect disk usage. If the disk usage of a node is high, resolve this problem by referring to **What If the Data Disk Capacity Is Insufficient?**

Constraints

- Only clusters of v1.19 or later allow reducing the capacity of the data disk used by container runtimes and kubelet.
- Only the EVS disk capacity can be adjusted. (Local disks are available only when the node specification is **disk-intensive** or **Ultra-high I/O**.)

Selecting a Data Disk

When selecting a data disk, consider the following factors:

- During image pull, the system downloads the image package (the .tar package) from the image repository, and decompresses the package. Then it deletes the package but retain the image file. During the decompression of the .tar package, the package and the decompressed image file coexist. Reserve the capacity for the decompressed files.
- Mandatory add-ons (such as everest and coredns) may be deployed on nodes during cluster creation. When calculating the data disk size, reserve about 2 GiB storage capacity for them.
- Logs are generated during application running. To ensure stable application running, reserve about 1 GiB storage capacity for each pod.

For details about the calculation formulas, see **OverlayFS** and **Device Mapper**.

OverlayFS

By default, the container engine and container image storage capacity of a node using the OverlayFS storage driver occupies 90% of the data disk capacity (you are advised to retain this value). All the 90% storage capacity is used for dockersys partitioning. The calculation methods are as follows:

- Capacity for storing container engines and container images requires 90% of the data disk capacity by default.
  - Capacity for dockersys volume (in the **/var/lib/docker** directory) requires 90% of the data disk capacity. The entire container engine and container image capacity (need 90% of the data disk capacity by default) are in the **/var/lib/docker** directory.
- Capacity for storing temporary kubelet and emptyDir requires 10% of the data disk capacity.

On a node using the OverlayFS, when an image is pulled, the .tar package is decompressed after being downloaded. During this process, the .tar package and the decompressed image file are stored in the dockersys volume, occupying about twice the actual image storage capacity. After the decompression is complete, the .tar package is deleted. Therefore, during image pull, after deducting the storage capacity occupied by the system add-on images, ensure that the remaining capacity of the dockersys volume is greater than twice the actual image storage capacity. To ensure that the containers can run stably, reserve certain capacity in the dockersys volume for container logs and other related files.

When selecting a data disk, consider the following formula:

Capacity of dockersys volume > Actual total image storage capacity x 2 + Total system add-on image storage capacity (about 2 GiB) + Number of

containers x Available storage capacity for a single container (about 1 GiB log storage capacity for each container)

NOTE

If container logs are output in the **json.log** format, they will occupy some capacity in the dockersys volume. If container logs are stored on persistent storage, they will not occupy capacity in the dockersys volume. Estimate the capacity of every container as required.

Example:

Assume that the node uses the OverlayFS and the data disk attached to this node is 20 GiB. According to **the preceding methods**, the capacity for storing container engines and images occupies 90% of the data disk capacity, and the capacity for the dockersys volume is 18 GiB (20 GiB x 90%). Additionally, mandatory add-ons may occupy about 2 GiB storage capacity during cluster creation. If you deploy a .tar package of 10 GiB, the package decompression takes 20 GiB of the dockersys volume's storage capacity. This, coupled with the storage capacity occupied by mandatory add-ons, exceeds the remaining capacity of the dockersys volume. As a result, the image pull may fail.

Device Mapper

By default, the capacity for storing container engines and container images of a node using the Device Mapper storage driver occupies 90% of the data disk capacity (you are advised to retain this value). The occupied capacity includes the dockersys volume and thinpool volume. The calculation methods are as follows:

- Capacity for storing container engines and container images requires 90% of the data disk capacity by default.
  - Capacity for the dockersys volume (in the **/var/lib/docker** directory) requires 20% of the capacity for storing container engines and container images.
  - Capacity forthe thinpool volume requires 80% of the container engine and container image storage capacity.
- Capacity for storing temporary kubelet and emptyDir requires 10% of the data disk capacity.

On a node using the Device Mapper storage driver, when an image is pulled, the .tar package is temporarily stored in the dockersys volume. After the .tar package is decompressed, the image file is stored in the thinpool volume, and the package in the dockersys volume will be deleted. Therefore, during image pull, ensure that the storage capacity of the dockersys volume and thinpool volume are sufficient, and note that the former is smaller than the latter. To ensure that the containers can run stably, reserve certain capacity in the dockersys volume for container logs and other related files.

When selecting a data disk, consider the following formulas:

- **Capacity for dockersys volume > Temporary storage capacity of the .tar package (approximately equal to the actual total image storage capacity) + Number of containers x Storage capacity of a single container (about 1 GiB log storage capacity must be reserved for each container)**
- **Capacity for thinpool volume > Actual total image storage capacity + Total add-on image storage capacity (about 2 GiB)**

> **NOTE**
>
> If container logs are output in the **json.log** format, they will occupy some capacity in the dockersys volume. If container logs are stored on persistent storage, they will not occupy capacity in the dockersys volume. Estimate the capacity of every container as required.

Example:

Assume that the node uses the Device Mapper and the data disk attached to this node is 20 GiB. According to **the preceding methods**, the container engine and image storage capacity occupies 90% of the data disk capacity, and the disk usage of the dockersys volume is 3.6 GiB. Additionally, the storage capacity of the mandatory add-ons may occupy about 2 GiB of the dockersys volume during cluster creation. The remaining storage capacity is about 1.6 GiB. If you deploy a .tar image package larger than 1.6 GiB, the storage capacity of the dockersys volume is insufficient for the package to be decompressed. As a result, the image pull may fail.

What If the Data Disk Capacity Is Insufficient?

Solution 1: Clearing images

Perform the following operations to clear unused images:

- Nodes that use containerd
  a. Obtain local images on the node.
     ```
     crictl images -v
     ```
  b. Delete the images that are not required by image ID.
     ```
     crictl rmi Image ID
     ```
- Nodes that use Docker
  a. Obtain local images on the node.
     ```
     docker images
     ```
  b. Delete the images that are not required by image ID.
     ```
     docker rmi Image ID
     ```

> **NOTE**
>
> Do not delete system images such as the cce-pause image. Otherwise, pods may fail to be created.

Solution 2: Expanding the disk capacity

Step 1 Expand the capacity of the data disk on the EVS console.

Step 2 Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More** > **Sync Server Data** in the row containing the target node.

Step 3 Log in to the target node.

Step 4 Run the **lsblk** command to check the block device information of the node.

A data disk is divided depending on the container storage **Rootfs**:

- Overlayfs: No independent thin pool is allocated. Image data is stored in the **dockersys** disk.
  ```
  # lsblk
  NAME            MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
  vda             8:0    0   50G  0 disk
  └─vda1          8:1    0   50G  0 part /
  ```

```
vdb                 8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker             # Space used by the container engine
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # Space used by Kubernetes
```

Run the following commands on the node to add the new disk capacity to the **dockersys** disk:

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

- Devicemapper: A thin pool is allocated to store image data.

```
# lsblk
NAME                         MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                          8:0    0   50G  0 disk
└─vda1                         8:1    0   50G  0 part /
vdb                          8:16   0  200G  0 disk
├─vgpaas-dockersys             253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta          253:1    0    3G  0 lvm
│ └─vgpaas-thinpool              253:3    0   67G  0 lvm                # Space used by thinpool
│   ...
├─vgpaas-thinpool_tdata          253:2    0   67G  0 lvm
│ └─vgpaas-thinpool              253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes              253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  - Run the following commands on the node to add the new disk capacity to the **thinpool** disk:

    ```
    pvresize /dev/vdb
    lvextend -l+100%FREE -n vgpaas/thinpool
    ```

  - Run the following commands on the node to add the new disk capacity to the **dockersys** disk:

    ```
    pvresize /dev/vdb
    lvextend -l+100%FREE -n vgpaas/dockersys
    resize2fs /dev/vgpaas/dockersys
    ```

----End

9.8 Analyzing Costs by Cluster

Background

The billing information of CCE is displayed based on the entire service by default. The costs of different clusters are not shown.

Solution

Add the **CCE-Cluster-ID** tag to the resources used in the clusters, obtain the costs by searching for this tag in cost center, and then analyze the costs by cluster to improve efficiency and cut costs.

Constraints

Generally, tags appear on the **Cost Tags** page 24 hours after resource expenditures are generated.

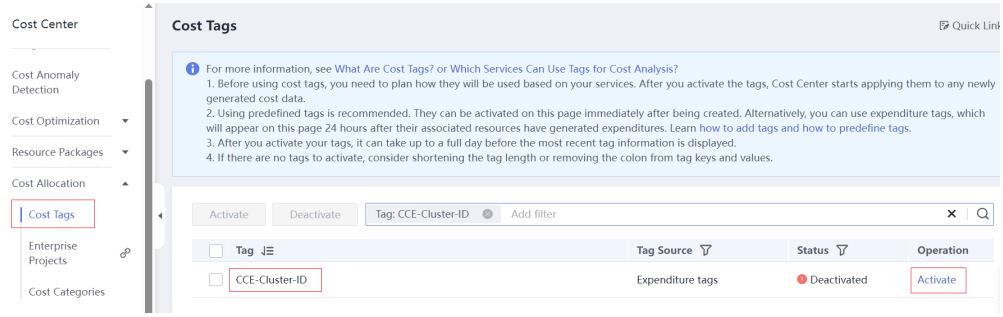
You can filter or group cost data by tag only after the tags are activated. If you activate the tags, they will be used to organize your resource costs generated thereafter.

Procedure

Step 1 Activate the **CCE-Cluster-ID** tag.

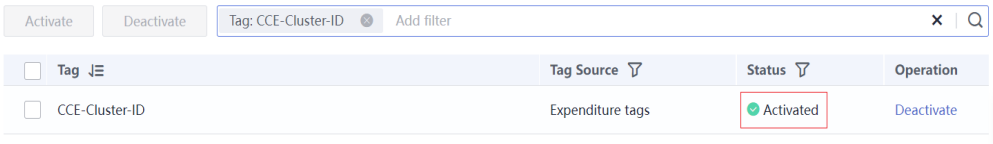
In **Cost Center**, choose **Cost Allocation** > **Cost Tags** in the navigation pane. Select **Tag** from the drop-down list for the filter criteria, search for **CCE-Cluster-ID**, and click **Activate** in the **Operation** column.

Figure 9-12 Activating the cost tag



The following information will be displayed.

Figure 9-13 Activation succeeded

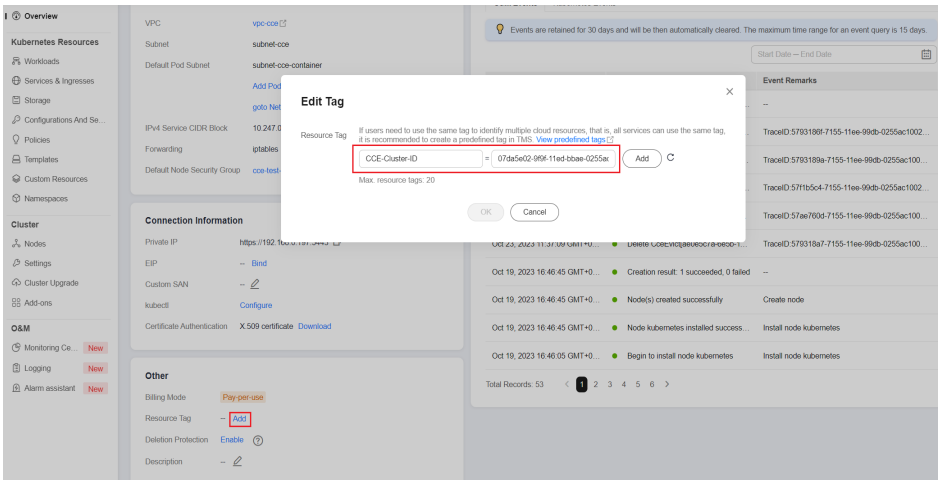


Step 2 Add the tag to resources used in a cluster.

Resources used in a cluster include master nodes, worker nodes, storage resources (such as EVS disks, SFS file systems, and OBS buckets), and network resources (such as load balancers and EIPs). By default, the **CCE-Cluster-ID** tag is added to the worker nodes.

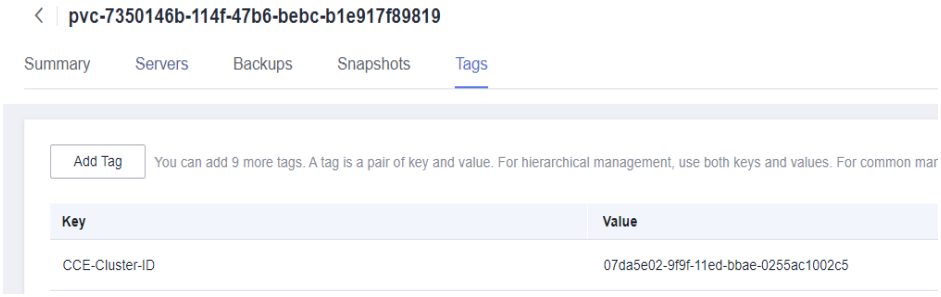
- Add the tag to a cluster.

  On the CCE console, click the name of the target cluster to access the cluster console. On the **Cluster Information** page, add the resource tag in the **Other** area.



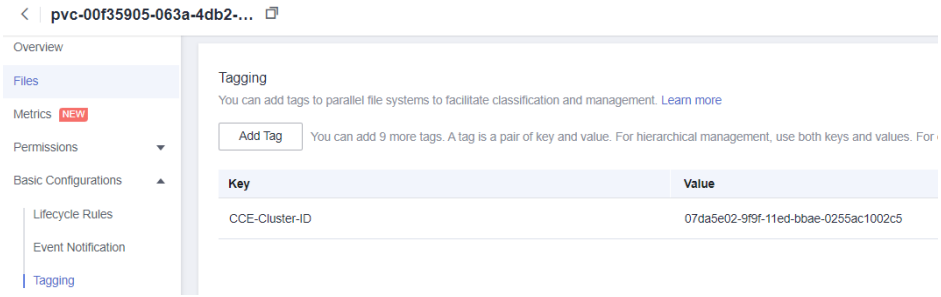
- Add the tag to an EVS disk.

  On the EVS console, click the name of the target disk to go to the details page. On the **Tags** tab, add a tag.



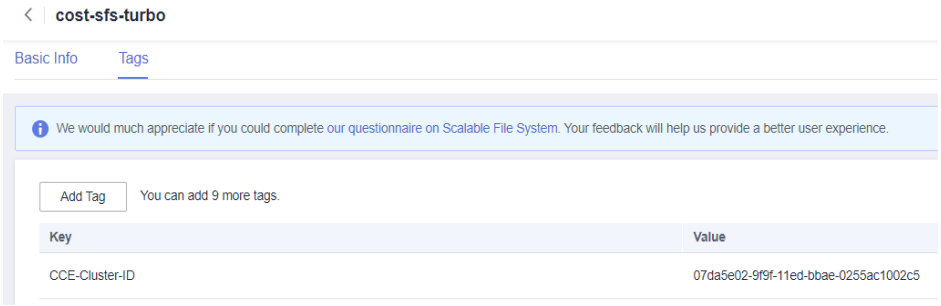
- Add the tag to an OBS bucket.

  On the OBS console, click the name of the target bucket to go to the details page. Choose **Basic Configurations** > **Tagging** and add a tag.



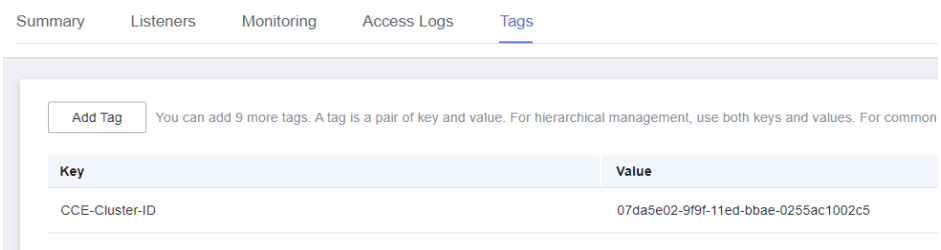
- Add the tag to an SFS Turbo file system.

  On the SFS console, click the name of the target SFS Turbo file system to go to the details page. On the page displayed, click the **Tags** tab and add a tag.



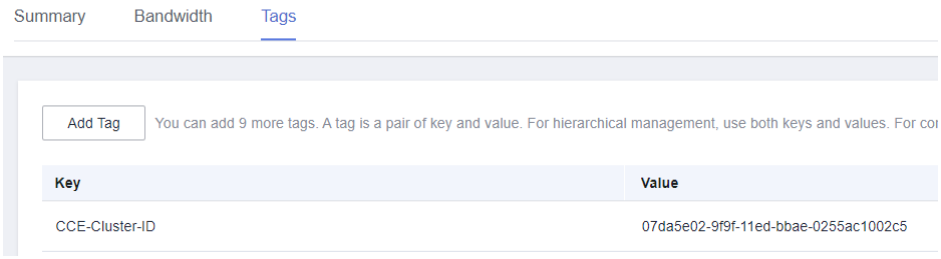
- Add the tag to a load balancer.

  On the ELB console, click the name of the target load balancer to go to the details page. On the page displayed, click the **Tags** tab and add a tag.



- Add the tag to an EIP.

On the EIP console, click the name of the target EIP to go to the details page. Click the **Tags** tab and add a tag.



Step 3 Analyze the cluster's costs.

On the **Cost Analysis** page, choose **CCE-Cluster-ID** and the ID of the desired cluster from the drop-down list.

Figure 9-14 Analyzing costs

Click **OK**.

Figure 9-15 Viewing the cluster's costs

Service Type	Accrued Total	Forecasted Total **	Aug 2023
Total cost(¥)	53.08	--	53.08
Elastic Cloud Serve...	30.44	--	30.44
Scalable File Servic...	17.50	--	17.50
Elastic Volume Ser...	5.14	--	5.14
Virtual Private Clou...	0.00	--	0.00
Object Storage Ser...	0.00	--	0.00
Elastic Load Balanc...	0.00	--	0.00

----End

9.9 Creating a CCE Turbo Cluster Using a Shared VPC

Shared VPC Overview

A shared VPC allows you to share your VPC resources with other accounts through the Resource Access Manager (RAM) service. For example, tenant A can share its VPC and subnets with tenant B. After tenant B accepted the sharing, tenant B can view the shared subnets and the shared VPC to which the shared subnets belong. Tenant B can use the shared subnets and VPC to create resources, such as CCE Turbo clusters. For details, see **VPC Sharing Overview**.

Application Scenarios

An enterprise organizes accounts in an orderly and centralized manner based on its organization structure or service form. Resources are managed in a unified manner and shared with other members to avoid repeated configurations. Unified security and O&M management makes it easy to configure and audit security policies.

For example, an enterprise IT account, the resource owner, creates a VPC and subnets and shares multiple subnets with other accounts.

- Account A is an enterprise service account and uses the shared subnet 1 to create resources.
- Account B is an enterprise service account and uses the shared subnet 2 to create resources.

Constraints

- Only CCE Turbo clusters support shared VPCs.
- Clusters created using a shared VPC do not support shared load balancers and NAT gateways.
- Clusters created using a shared VPC do not support SFS, OBS, and SFS Turbo storage volumes.
- If a CCE Turbo cluster has been created using a shared VPC, the owner of the shared VPC should not turn off the VPC sharing. Otherwise, the CCE Turbo cluster will malfunction.

Procedure

After account A shares a VPC with account B, account B can select the shared VPC and shared subnets when creating a CCE Turbo cluster.

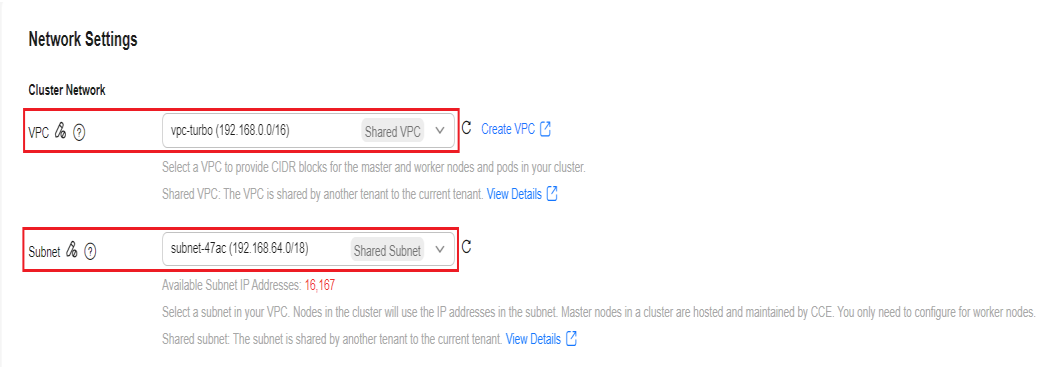
Step 1 (For account A) Use RAM to create a shared VPC and specify account B as the resource user. For details, see **Creating a Resource Share**.

After the resource sharing is created, RAM sends an invitation to account B. Account B can access and use the shared resources only after accepting the invitation.

Step 2 (For account B) Log in to the CCE console and create a CCE Turbo cluster.

Select the VPC shared by account A when configuring network for the cluster. For details about other configurations, see **Buying a CCE Cluster**.

Figure 9-16 Selecting a shared VPC



----End

10 Networking

10.1 Planning CIDR Blocks for a Cluster

Before creating a cluster on CCE, determine the number of VPCs, number of subnets, container CIDR blocks, and Services for access based on service requirements.

This topic describes the addresses in a CCE cluster in a VPC and how to plan CIDR blocks.

Constraints

To access a CCE cluster through a VPN, ensure that the VPN does not conflict with the VPC CIDR block where the cluster resides and the container CIDR block.

Basic Concepts

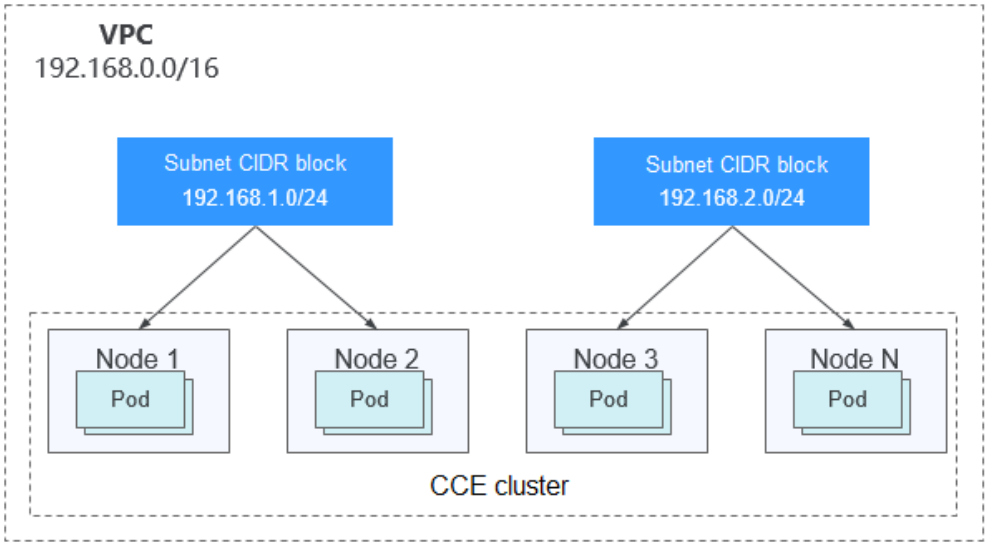
- **VPC CIDR Block**

  Virtual Private Cloud (VPC) enables you to provision logically isolated, configurable, and manageable virtual networks for cloud servers, cloud containers, and cloud databases. You have complete control over your virtual network, including selecting your own CIDR block, creating subnets, and configuring security groups. You can also assign EIPs and allocate bandwidth in your VPC for secure and easy access to your business system.

- **Subnet CIDR Block**

  A subnet is a network that manages ECS network planes. It supports IP address management and DNS. The IP addresses of all ECSs in a subnet belong to the subnet.

Figure 10-1 VPC CIDR block architecture



By default, ECSs in all subnets of the same VPC can communicate with one another, while ECSs in different VPCs cannot communicate with each other.

You can create a peering connection on VPC to enable ECSs in different VPCs to communicate with each other.

- **Container (Pod) CIDR Block**

  Pod is a Kubernetes concept. Each pod has an IP address.

  When creating a cluster on CCE, you can specify the pod (container) CIDR block, which cannot overlap with the subnet CIDR block. For example, if the subnet CIDR block is 192.168.0.0/16, the container CIDR block cannot be 192.168.0.0/18 or 192.168.1.0/18, because these addresses are included in 192.168.0.0/16.

- **Container Subnet** (Only for CCE Turbo Clusters)

  In a CCE Turbo cluster, a container is assigned an IP address from the CIDR block of a VPC. The container subnet can overlap with the subnet CIDR block. Note that the subnet you select determines the maximum number of pods in the cluster. After a cluster is created, you can only add container subnets but cannot delete them.

- **Service CIDR Block**

  Service is also a Kubernetes concept. Each Service has an address. When creating a cluster on CCE, you can specify the Service CIDR block. Similarly, the Service CIDR block cannot overlap with the subnet CIDR block or the container CIDR block. The Service CIDR block can be used only within a cluster.

Single-VPC Single-Cluster Scenarios

CCE Clusters: include clusters in VPC network model and container tunnel network model. **Figure 10-2** shows the CIDR block planning of a cluster.

- VPC CIDR Block: specifies the VPC CIDR block where the cluster resides. The size of this CIDR block affects the maximum number of nodes that can be created in the cluster.

- Subnet CIDR Block: specifies the subnet CIDR block where the node in the cluster resides. The subnet CIDR block is included in the VPC CIDR block. Different nodes in the same cluster can be allocated to different subnet CIDR blocks.
- Container CIDR Block: cannot overlap with the subnet CIDR block.
- Service CIDR Block: cannot overlap with the subnet CIDR block or the container CIDR block.

Figure 10-2 Network CIDR block planning in single-VPC single-cluster scenarios (CCE cluster)

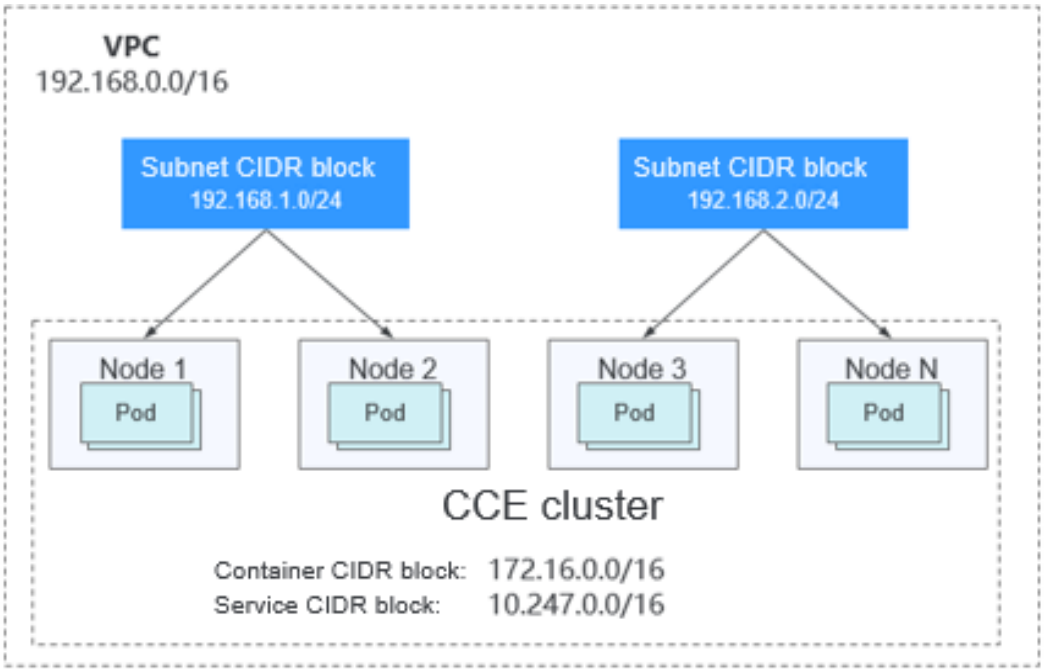
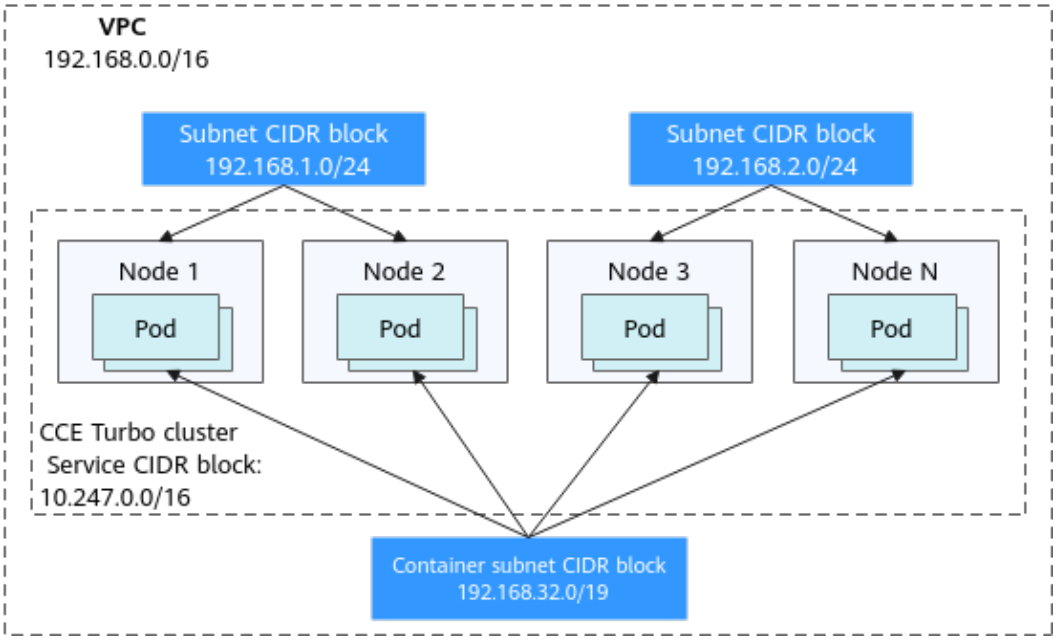


Figure 10-3 shows the CIDR block planning for a **CCE Turbo cluster** (Cloud Native Network 2.0).

- VPC CIDR Block: specifies the VPC CIDR block where the cluster resides. The size of this CIDR block affects the maximum number of nodes that can be created in the cluster.
- Subnet CIDR Block: specifies the subnet CIDR block where the node in the cluster resides. The subnet CIDR block is included in the VPC CIDR block. Different nodes in the same cluster can be allocated to different subnet CIDR blocks.
- Container Subnet CIDR Block: The container subnet is included in the VPC CIDR block and can overlap with the subnet CIDR block or even be the same as the subnet CIDR block. Note that the container subnet size determines the maximum number of containers in the cluster because IP addresses in the VPC are directly allocated to containers. After a cluster is created, you can only add container subnets but cannot delete them. Set a larger IP address segment for the container subnet to prevent insufficient container IP addresses.
- Service CIDR Block: cannot overlap with the subnet CIDR block or the container CIDR block.

Figure 10-3 Network CIDR block planning in single-VPC single-cluster scenarios (CCE Turbo Clusters)



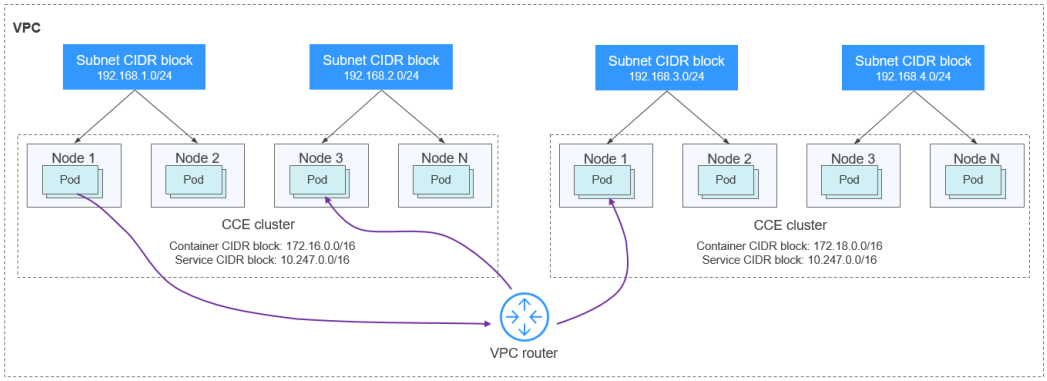
Single-VPC Multi-Cluster Scenarios

VPC network model

Pod packets are forwarded through VPC routes. CCE automatically configures a routing table on the VPC routes to each container CIDR block. The network scale is limited by the VPC route table. **Figure 10-4** shows the CIDR block planning of the cluster.

- VPC CIDR Block: specifies the VPC CIDR block where the cluster resides. The size of this CIDR block affects the maximum number of nodes that can be created in the cluster.
- Subnet CIDR Block: The subnet CIDR block in each cluster cannot overlap with the container CIDR block.
- Container CIDR Block: If multiple VPC network model clusters exist in a single VPC, the container CIDR blocks of all clusters cannot overlap because the clusters use the same routing table. In this case, if the node security group allows container CIDR block from the peer cluster, pods in one cluster can directly access pods in another cluster through the pod IP addresses.
- Service CIDR Block: can be used only in clusters. Therefore, the Service CIDR blocks of different clusters can overlap, but cannot overlap with the subnet CIDR block and container CIDR block of the cluster.

Figure 10-4 VPC network - multi-cluster scenario

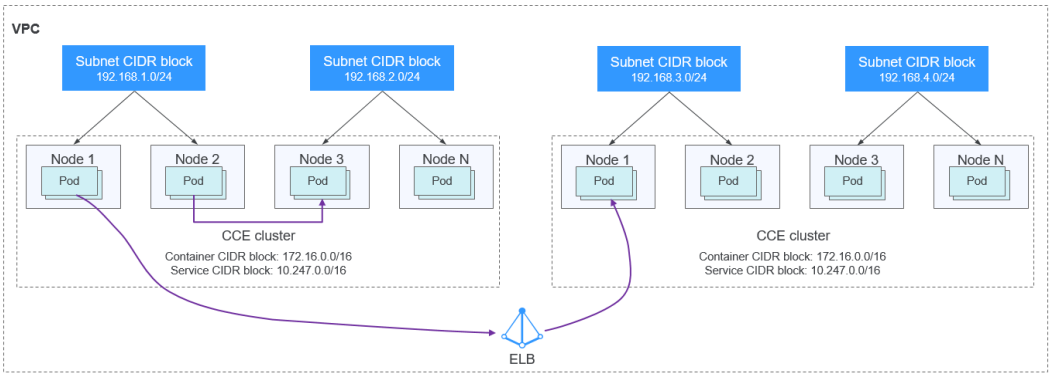


Tunnel network model

Though at some cost of performance, the tunnel encapsulation enables higher interoperability and compatibility with advanced features (such as network policy-based isolation), meeting the requirements of most applications. **Figure 10-5** shows the CIDR block planning of the cluster.

- VPC CIDR Block: specifies the VPC CIDR block where the cluster resides. The size of this CIDR block affects the maximum number of nodes that can be created in the cluster.
- Subnet CIDR Block: The subnet CIDR block in each cluster cannot overlap with the container CIDR block.
- Container CIDR Block: The container CIDR blocks of all clusters can overlap. In this case, pods in different clusters cannot be directly accessed through pod IP addresses. Pods in different clusters need to access each other through Services. The LoadBlancer Services are recommended.
- Service CIDR Block: can be used only in clusters. Therefore, the Service CIDR blocks of different clusters can overlap, but cannot overlap with the subnet CIDR block and container CIDR block of the cluster.

Figure 10-5 Tunnel network - multi-cluster scenario

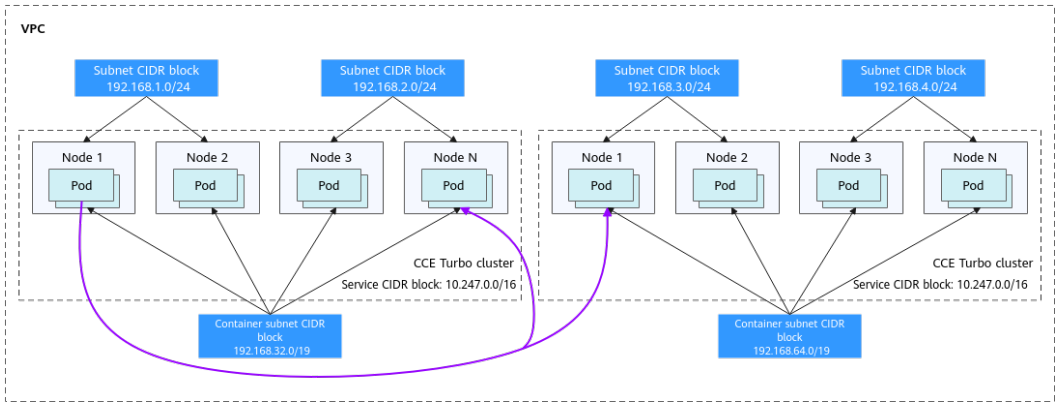


Cloud Native 2.0 network model (CCE Turbo Clusters)

In this mode, container IP addresses are allocated from the VPC CIDR block. ELB passthrough networking is supported to direct access requests to containers. Security groups and multiple types of VPC networks can be bound to deliver high performance.

- VPC CIDR Block: specifies the VPC CIDR block where the cluster resides. In a CCE Turbo cluster, the CIDR block size affects the total number of nodes and containers that can be created in the cluster.
- Subnet CIDR Block: There is no special restriction on the subnet CIDR blocks in CCE Turbo clusters.
- Container Subnet: The CIDR block of the container subnet is included in the VPC CIDR block. Container subnets in different clusters can overlap with each other or overlap with the subnet CIDR block. However, you are advised to stagger the container CIDR blocks of different clusters and ensure that the container subnet CIDR blocks have sufficient IP addresses. In this case, if the ENI security group of the cluster allows the container CIDR block of the peer cluster, pods in different clusters can directly access each other through IP addresses.
- Service CIDR Block: can be used only in clusters. Therefore, the Service CIDR blocks of different clusters can overlap, but cannot overlap with the subnet CIDR block and container subnet CIDR block of the cluster.

Figure 10-6 Cloud Native 2.0 network - multi-cluster scenario



Clusters using different networks

When a VPC contains clusters created with different network models, comply with the following rules when creating a cluster:

- VPC CIDR Block: In this scenario, all clusters are located in the same VPC CIDR block. Ensure that there are sufficient available IP addresses in the VPC.
- Subnet CIDR Block: Ensure that the subnet CIDR block does not overlap with the container CIDR block. Even in some scenarios (for example, coexistence with CCE Turbo clusters), the subnet CIDR block can overlap with the container (subnet) CIDR block. However, this is not recommended.
- Container CIDR Block: Ensure that the container CIDR blocks of clusters in **VPC network model** do not overlap.
- Service CIDR Block: The Service CIDR blocks of all clusters can overlap, but cannot overlap with the subnet CIDR block and container CIDR block of the cluster.

Cross-VPC Cluster Interconnection

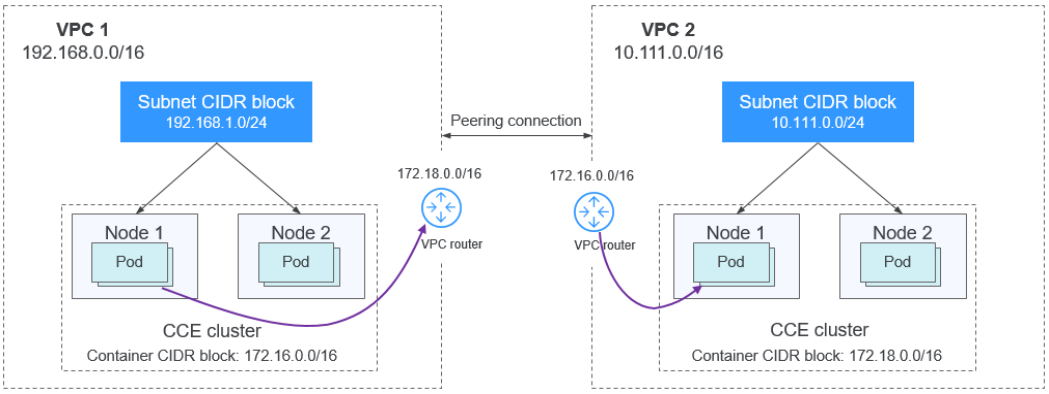
If VPCs cannot communicate with each other, a VPC peering connection is used to ensure communication between VPCs. When two VPC networks are

interconnected, you can configure the packets to be sent to the peer VPC in the route table. For details, see **VPC Peering Connection Overview**.

Clusters using VPC networks

To allow clusters that use VPC networks to access each other across VPCs, add routes to the two ends of the VPC peering after a VPC peering connection is created.

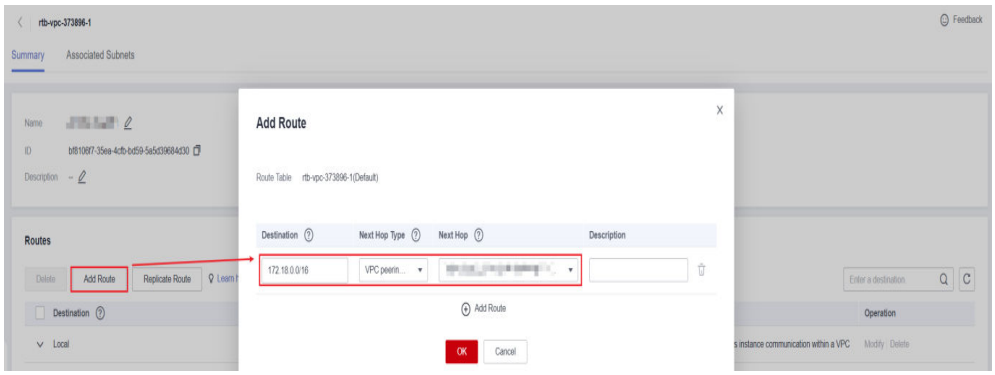
Figure 10-7 VPC network - VPC interconnection scenario



When creating a VPC peering connection between containers across VPCs, pay attention to the following points:

- The VPC to which the clusters belong must not overlap. In each cluster, the subnet CIDR block cannot overlap with the container CIDR block.
- The container CIDR blocks of clusters at both ends cannot overlap, but the Service CIDR blocks can.
- If the request end cluster uses the VPC network, check whether the node security group in the destination cluster allows the container CIDR block of the request end cluster. If yes, pods in one cluster can directly access pods in another cluster through the pod IP address. Similarly, if nodes running in the clusters at the two ends of the VPC peering connection need to access each other, the node security group must allow the VPC CIDR block of the peer cluster.
- You need to add routes for accessing the peer network CIDR block to the VPC routing tables at both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2.
  - **Add the VPC CIDR block of the peer cluster**: After the route of the VPC CIDR block is added, a pod in a cluster can access another cluster node. For example, the pod can access the port of a NodePort Service.
  - **Add peer container CIDR block**: After the route of the container CIDR block is added, a pod can directly access pods in another cluster through the container IP addresses.

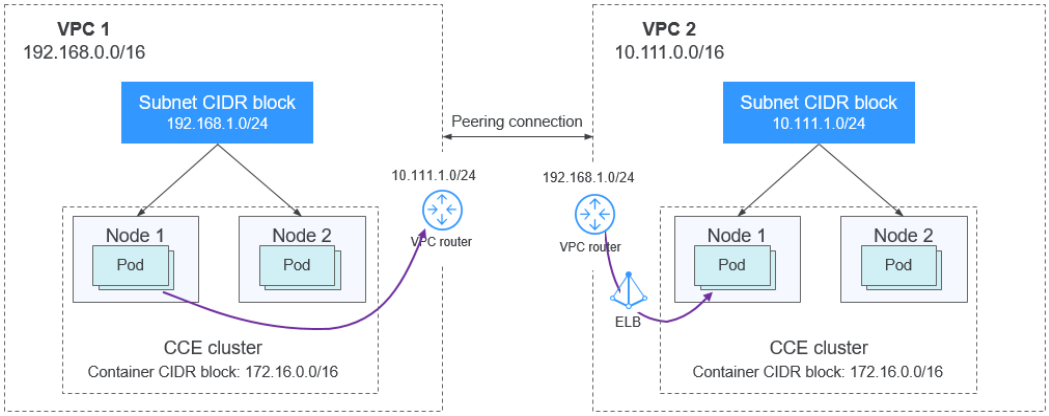
Figure 10-8 Adding the peer container CIDR block to the local route on the VPC console



Clusters using tunnel networks

To allow clusters that use tunnel networks to access each other across VPCs, add routes to the two ends of the VPC peering after a VPC peering connection is created.

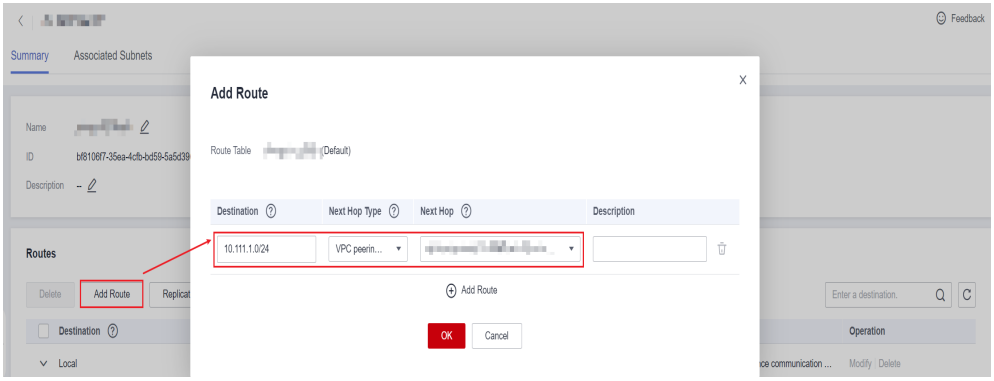
Figure 10-9 Tunnel network - VPC interconnection scenario



Pay attention to the following:

- The VPCs of the peer clusters must not overlap.
- The container CIDR blocks of all clusters can overlap, so do the Service CIDR blocks.
- If the request end cluster uses the tunnel network, check whether the node security group in the destination cluster allows the VPC CIDR block (including the node subnets) of the request end cluster. If yes, nodes in one cluster can access nodes in another cluster. However, pods in different clusters cannot be directly accessed using pod IP addresses. Access between pods in different clusters requires Services. The LoadBlancer Services are recommended.
- The VPC CIDR block route of the peer cluster must be added to the VPC routing tables of both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2. After the route of the VPC CIDR block is added, the pod can access another cluster node, for example, accessing the port of a NodePort Service.

Figure 10-10 Adding the subnet CIDR block of the peer cluster node to the local route on the VPC console



Clusters using Cloud Native 2.0 networks (CCE Turbo clusters)

After creating a VPC peering connection, add routes of the VPC peering connection to the both ends so that the two VPCs can communicate with each other. Pay attention to the following:

- The VPCs of the clusters at the two ends must not overlap.
- If the request end cluster uses the Cloud Native 2.0 network, check whether the ENI security group (named in the format of *{Cluster name}*-**cce-eni-***{Random ID}*) of the destination cluster allows the VPC CIDR block (including the node subnets and container CIDR block) of the request end cluster. If yes, pods in one cluster can directly access pods in another cluster through the pod IP addresses. Similarly, if nodes in the clusters at the two ends of the VPC peering need to access each other, allow the VPC CIDR block of the peer cluster in the node security group (named in the format of *{Cluster name}*-**cce-node-***{Random ID}*).
- The VPC CIDR block route of the peer cluster must be added to the VPC routing tables of both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2. After the route of the VPC CIDR block is added, the pod can access pod IP addresses or nodes in another cluster.

Clusters using different networks

If clusters using different networks need to communicate with each other across VPCs, every one of them may serve as the request end or destination end. Pay attention to the following:

- The VPC CIDR block to which the cluster belongs cannot overlap with the VPC CIDR block of the peer cluster.
- Cluster subnet CIDR blocks cannot overlap with the container CIDR blocks.
- Container CIDR blocks in different clusters cannot overlap with each other.
- If pods or nodes in different clusters need to access each other, the security groups of the clusters on both ends must allow the corresponding CIDR blocks based on the following rules:
  - If the request end cluster uses the VPC network, the node security group of the destination cluster must allow the VPC CIDR block (including the node subnets and container CIDR block) of the request end cluster.

  - If the request end cluster uses the tunnel network, the node security group of the destination cluster must allow the VPC CIDR block (including the node subnets) of the request end cluster.
  - If the request end cluster uses the Cloud Native 2.0 network, the ENI security group and node security group of the destination cluster must allow the VPC CIDR block (including node subnets and container CIDR block) of the request end cluster.
- The VPC CIDR block route of the peer cluster must be added to the VPC routing tables of both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2. After the route of the VPC CIDR block is added, the pod can access another cluster node, for example, accessing the port of a NodePort Service.

  If a cluster uses the VPC network, the VPC routing tables at both ends must contain its container CIDR block. After the container CIDR block route is added, the pod can directly access pods in another cluster through the container IP addresses.

VPC-IDC Scenarios

Similar to the VPC interconnection scenario, some CIDR blocks in the VPC are routed to the IDC. The pod IP addresses of CCE clusters cannot overlap with the addresses within these CIDR blocks. To access the pod IP addresses in the cluster in the IDC, configure the route table to the private line VBR on the IDC.

10.2 Selecting a Network Model

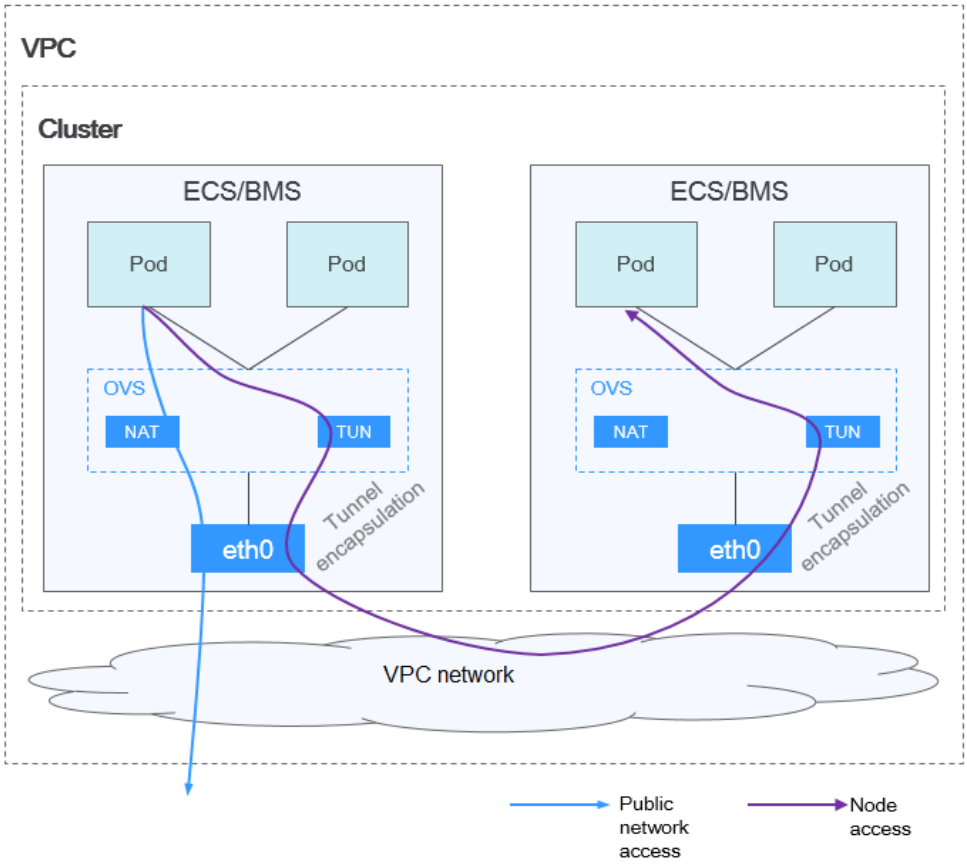
CCE uses proprietary, high-performance container networking add-ons to support the tunnel network, Cloud Native 2.0 network, and VPC network models.

CAUTION

After a cluster is created, the network model cannot be changed. Exercise caution when selecting a network model.

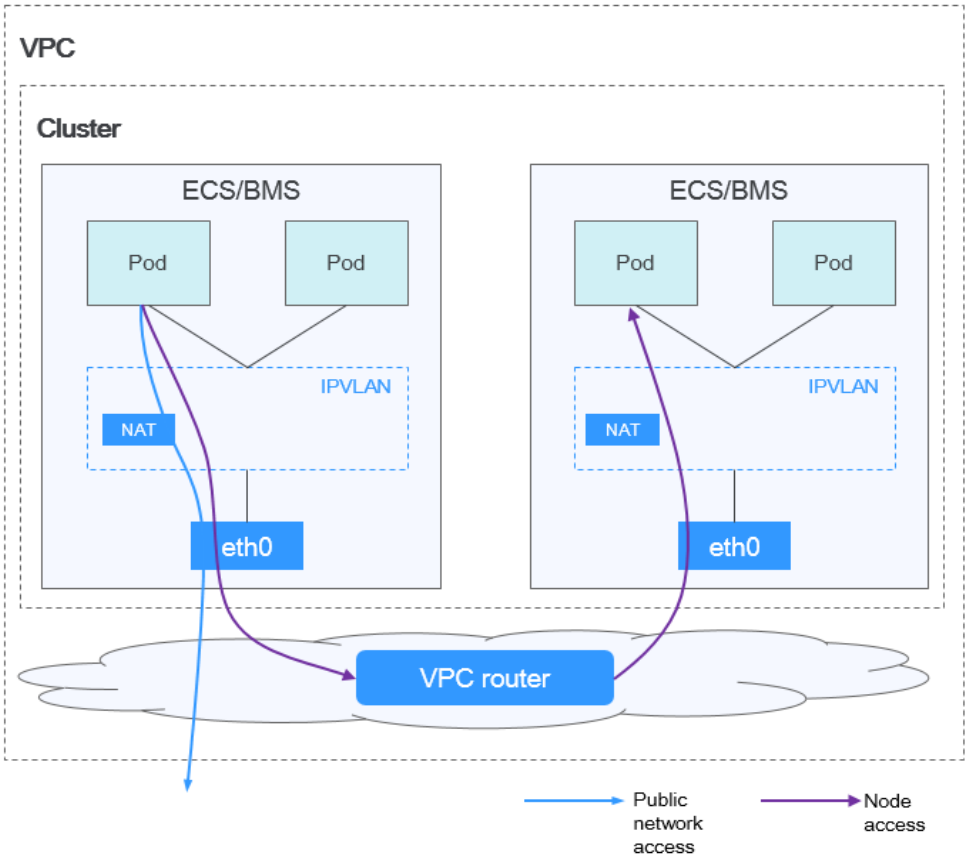
- **Tunnel network**: The container network is an overlay tunnel network on top of a VPC network and uses the VXLAN technology. This network model is applicable when there is no high requirements on performance. VXLAN encapsulates Ethernet packets as UDP packets for tunnel transmission. Though at some cost of performance, the tunnel encapsulation enables higher interoperability and compatibility with advanced features (such as network policy-based isolation), meeting the requirements of most applications.

Figure 10-11 Container tunnel network



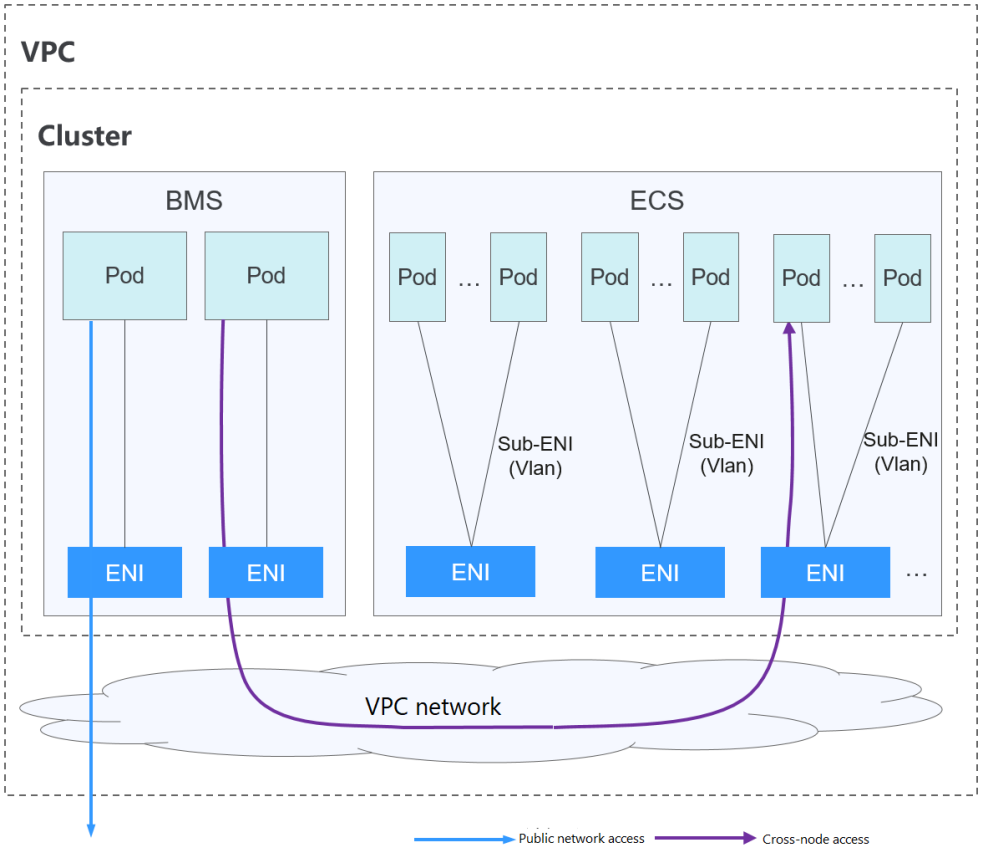
- **VPC network**: The container network uses VPC routing to integrate with the underlying network. This network model is applicable to performance-intensive scenarios. The maximum number of nodes allowed in a cluster depends on the route quota in a VPC network. Each node is assigned a CIDR block of a fixed size. VPC networks are free from tunnel encapsulation overhead and outperform container tunnel networks. In addition, as VPC routing includes routes to node IP addresses and container network segment, container pods in the cluster can be directly accessed from outside the cluster.

Figure 10-12 VPC network



- **Cloud Native Network 2.0**: The container network deeply integrates the elastic network interface (ENI) capability of VPC, uses the VPC CIDR block to allocate container addresses, and supports passthrough networking to containers through a load balancer.

Figure 10-13 Cloud Native 2.0 network



The following table lists the differences between the network models.

Table 10-1 Network model comparison

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Application scenarios	• Common container service scenarios • Scenarios that do not have high requirements on network latency and bandwidth	• Scenarios that have high requirements on network latency and bandwidth • Containers can communicate with VMs using a microservice registration framework, such as Dubbo and CSE.	• Scenarios that have high requirements on network latency, bandwidth, and performance • Containers can communicate with VMs using a microservice registration framework, such as Dubbo and CSE.
Core technology	OVS	IPvlan and VPC route	VPC ENI/sub-ENI

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Applicable clusters	CCE standard cluster	CCE standard cluster	CCE Turbo cluster
Network isolation	Kubernetes native NetworkPolicy for pods	No	Pods support security group isolation.
Passthrough networking	No	No	Yes
IP address management	• The container CIDR block is allocated separately. • CIDR blocks are divided by node and can be dynamically allocated (CIDR blocks can be dynamically added after being allocated.)	• The container CIDR block is allocated separately. • CIDR blocks are divided by node and statically allocated (the CIDR block cannot be changed after a node is created).	The container CIDR block is divided from the VPC subnet and does not need to be allocated separately.
Network performance	Performance loss due to VXLAN encapsulation	No tunnel encapsulation. Cross-node packets are forwarded through VPC routers, delivering performance equivalent to that of the host network.	The container network is integrated with the VPC network, eliminating performance loss.

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Networking scale	A maximum of 2000 nodes are supported.	Suitable for small- and medium-scale networks due to the limitation on VPC routing tables. It is recommended that the number of nodes be less than or equal to 1000. Each time a node is added to the cluster, a route is added to the VPC routing tables (including the default and custom ones). Therefore, the cluster scale is limited by the VPC routing tables. For details about routing tables, see **Constraints**.	A maximum of 2000 nodes are supported.

NOTICE

1. The scale of a cluster that uses the VPC network model is limited by the custom routes of the VPC. Therefore, you need to estimate the number of required nodes before creating a cluster.
2. By default, VPC routing network supports direct communication between containers and hosts in the same VPC. If a peering connection policy is configured between the VPC and another VPC, the containers can directly communicate with hosts on the peer VPC. In addition, in hybrid networking scenarios such as Direct Connect and VPN, communication between containers and hosts on the peer end can also be achieved with proper planning.

10.3 Allowing Containers and IDCs to Communicate with Each Other Through VPC and Direct Connect

Application Scenarios

With VPC and Direct Connect, the container CIDR block (172.56.0.0/16) and IDC CIDR block (10.1.123.0/24) can communicate with each other in the cluster using the VPC network model.

Figure 10-14 Example network topology

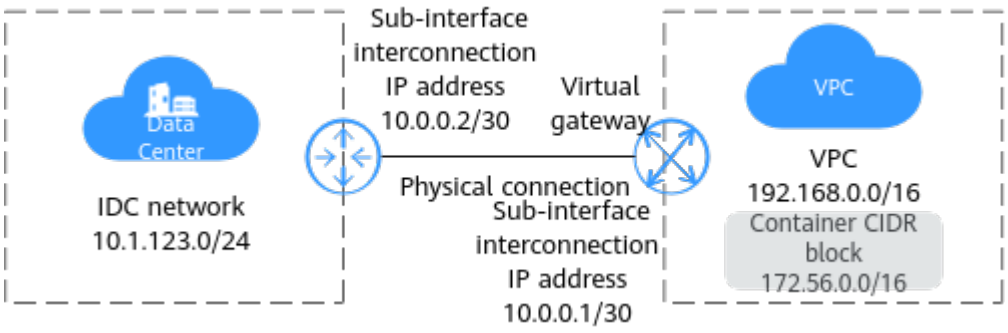


Table 10-2 Address information

Networking	CIDR Block
User's IDC network	10.1.123.0/24
Remote and local gateways (addresses for interconnection)	Huawei Cloud: 10.0.0.1/30 User: 10.0.0.2/30
VPC	192.168.0.0/16
Container CIDR block	172.56.0.0/16

Prerequisites

An IDC is available, and the Direct Connect service has been applied for.

Procedure

Step 1 Create a connection.

1. Log in to the management console, click in the upper left corner, and select the desired region and project. Click at the upper left corner and choose **Networking** > **Direct Connect** in the expanded list.
2. In the navigation pane on the left of the console, choose **Direct Connect** > **Connections**. On the displayed page, click **Create Connection**.
3. On the **Create Connection** page, click **Self Service Installation**.

   On the **Create Connection** page, enter the equipment room details and select the Direct Connect location and port based on **Table 10-3**.

   Table 10-3 Parameters for purchasing a connection

Parameter	Description
Billing Mode	Specifies how you are charged. Currently, only **Yearly/Monthly** is supported.

Parameter	Description
Region	Specifies the region where the connection is deployed. You can change the region in the upper left corner of the console.
Connection Name	Specifies the name of your connection.
Location	Specifies the Direct Connect location where your leased line can be connected to.
Carrier	Specifies the carrier that provides the leased line.
Port Type	Specifies the type of the port used by the connection. There are four types of ports: 1GE, 10GE, 40GE, and 100GE.
Leased Line Bandwidth	Specifies the bandwidth of the connection in the unit of Mbit/s. This is the bandwidth of the leased line you have purchased from the carrier.
Equipment Room Address	Specifies the address of your equipment room. The address must be specific to the floor your equipment room is on, for example, XX Equipment Room, XX Building, No. XX, Huajing Road, Pudong District, Shanghai.
Tag	Identifies the connection. A tag consists of a key and a value. You can add 10 tags to a connection. Tag keys and values must meet the requirements listed in **Table 10-4**. **NOTE** If a predefined tag has been created on TMS, you can directly select the corresponding tag key and value. For details about predefined tags, see **Predefined Tag Overview**.
Description	Provides supplementary information about the connection.
Contact Person/ Phone Number/ Email	Specifies information about the person who is responsible for your connection. If the contact information is not provided, your account information will be queried. This will prolong the review period.
Required Duration	Specifies how long the connection will be used.
Auto-renew	Specifies whether to automatically renew the connection to ensure service continuity. It is recommended that you set the auto-renewal period to be the same as the required duration. If the required duration is three months, the system automatically renews the subscription for every three months.

Parameter	Description
Enterprise Project	Centrally manages cloud resources and members by project.

Table 10-4 Tag key and value requirements

Parameter	Requirements
Key	– Cannot be left blank. – Must be unique for each resource. – Can contain a maximum of 36 characters. – Can contain only letters, digits, hyphens, underscores, and Unicode characters from \u4e00 to \u9fff.
Value	– Can be left blank. – Can contain a maximum of 43 characters. – Can contain only letters, digits, period, hyphens, underscores, and Unicode characters from \u4e00 to \u9fff.

4. Click **Confirm Configuration**
5. Confirm the order and click **Pay Now**.
6. Click **Pay Now**.

Step 2 Create a virtual gateway.

1. Choose **Direct Connect** > **Virtual Gateways**, and click **Create Virtual Gateway** on the right. Add the VPC CIDR block and the container CIDR block in the VPC network model.

Figure 10-15 Creating a virtual gateway

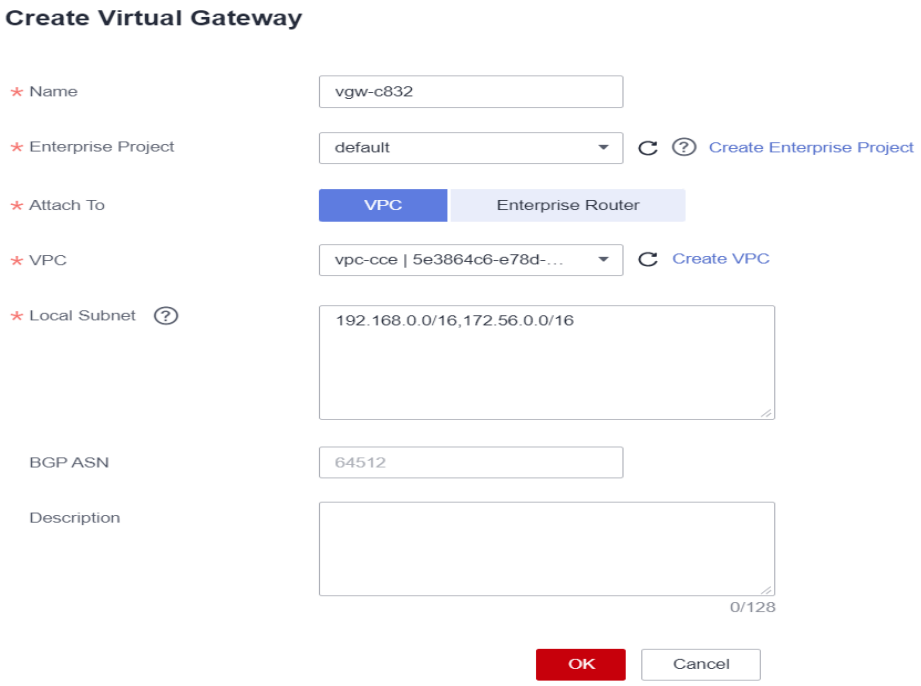


Table 10-5 Virtual gateway parameters

Parameter	Description
Name	Specifies the virtual gateway name. You can enter 1 to 64 characters.
Enterprise Project	Centrally manages cloud resources and members by project.
Attach To	Select **VPC**.
VPC	Specifies the VPC you want to access using the connection.
Local Subnet	Specifies the CIDR blocks of subnets in the VPC to connect to the on-premises network. In this example, the cluster uses the VPC network model. Enter the VPC CIDR block (192.168.0.0/16) and container CIDR block (172.56.0.0/16). For clusters using the container tunnel network and Cloud Native 2.0 Network models, you only need to enter the VPC CIDR block.
Description	Provides supplementary information about the virtual gateway. The description can contain a maximum of 128 characters.

2. Click **OK**.

   When the virtual gateway status changes **Normal**, the virtual gateway has been created.

Step 3 Create a virtual interface.

1. Choose **Direct Connect** > **Virtual Interfaces**, and click **Create Virtual Interface** on the right.
2. Configure the parameters based on **Table 10-6**.

Figure 10-16 Creating a virtual interface

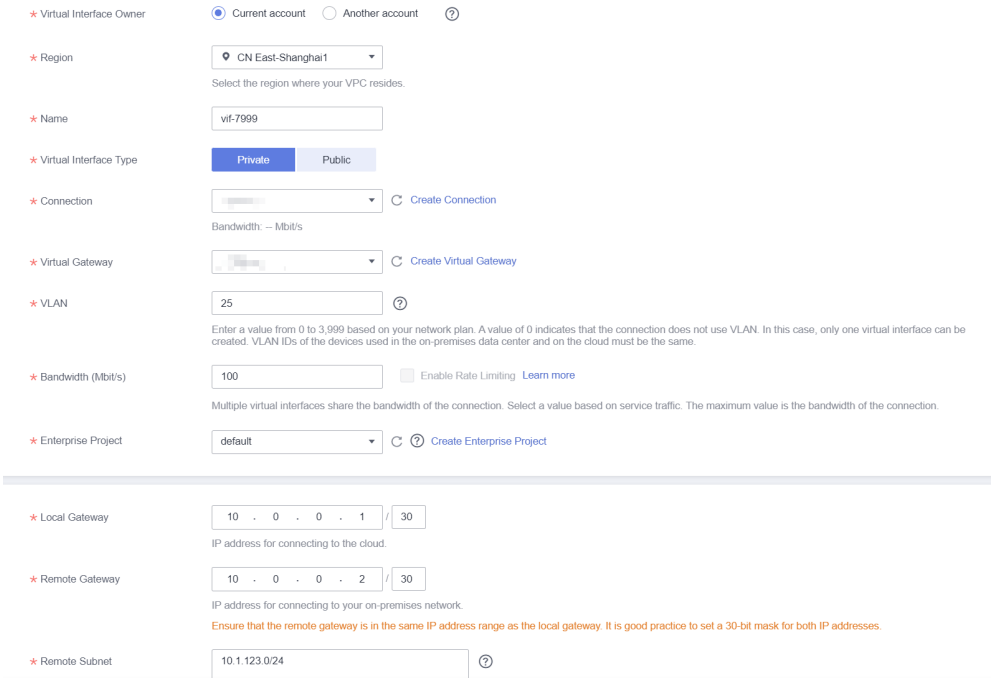


Table 10-6 Parameters required for creating a virtual interface

Parameter	Description
Region	Specifies the region where the connection is deployed. You can change the region in the upper left corner of the console.
Name	Specifies the virtual interface name. You can enter 1 to 64 characters.
Connection	Specifies the connection you use to connect your data center to the cloud.
Virtual Gateway	Specifies the virtual gateway that the virtual interface connects to.

Parameter	Description
VLAN	Specifies the VLAN of the virtual interface. Configure the VLAN if you create a connection on your own. The VLAN for a hosted connection will be allocated by the carrier or partner. You do not need to configure the VLAN.
Bandwidth	Specifies the bandwidth that can be used by the virtual interface in the unit of Mbit/s. The bandwidth cannot exceed that of the connection.
Enterprise Project	Centrally manages cloud resources and members by project.
Local Gateway	Specifies the IP address for connecting to the cloud. In this example, the IP address is **10.0.0.1/30**.
Remote Gateway	Specifies the IP address for connecting to your on-premises network. The remote gateway must be in the same IP address range as the local gateway. Generally, a subnet with a 30-bit mask is recommended. In this example, the IP address is **10.0.0.2/30**.
Remote Subnet	Specifies the subnets of your on-premises network. If multiple remote subnets are available, use commas (,) to separate them. In this example, the IP address is **10.1.123.0/24**.
Routing Mode	Specifies the routing mode. Two options are available, **Static** and **BGP**. If there are two or more connections, select BGP routing.
BGP ASN	Specifies the ASN of the BGP peer. This parameter is mandatory when you select BGP routing.

Parameter	Description
BGP MD5 Authentication Key	Specifies the password used to authenticate the BGP peer using MD5. This parameter is mandatory when BGP routing is selected, and the parameter values on both gateways must be the same. The key contains 8 to 255 characters and must contain at least two types of the following characters: – Uppercase letters – Lowercase letters – Digits – Special characters ~!, .:;-_"(){}[]/@#$ %^&*+|=
Description	Provides supplementary information about the virtual interface.

3. Click **Submit**. When the status of the virtual interface changes **Normal**, the virtual interface has been created.
4. Ping the IP address of a server in the VPC from your data center to test network connectivity.

   Now your environment can connect to the cloud and access the desired VPC.

   NOTE

   After creating a virtual interface, configure your devices and security group rules to allow access on and off the cloud.

Step 4 Test the connectivity.

1. Run the **traceroute** command to check whether the IDC host can communicate with the container.
   a. If the route is normal, Direct Connect has a return route.
   b. If the IDC route to the cloud gateway of Direct Connect is abnormal, check whether the route settings at both ends of Direct Connect are correct.
2. If the IP address cannot be tracerouted, try the ping or telnet operation. Before pinging the address, ensure that the ICMP policy has been enabled for the security group if the target is an ECS.

----End

10.4 Enabling a CCE Cluster to Resolve Domain Names on Both On-Premises IDCs and HUAWEI CLOUD

10.4.1 Solution Overview

Issues

Microservices are increasingly used to deploy applications. When microservices access each other, they need to resolve domain names.

When you have on-premises IDCs with internal domain names configured, and you have deployed containerized applications on both these IDCs and cloud, you need to enable the containers and nodes in CCE clusters to resolve domain names of both the IDC and cloud.

Suppose you have reconstructed one of your applications using microservices. You run the application management backend in a CCE cluster, deploy the content moderation service in the on-premises IDC, and use the image recognition service on Huawei Cloud. The VPC where CCE resides is connected to the IDC through a private line. **Figure 10-17** shows the deployment architecture.

When a user accesses this application, the following interaction is involved between different microservices:

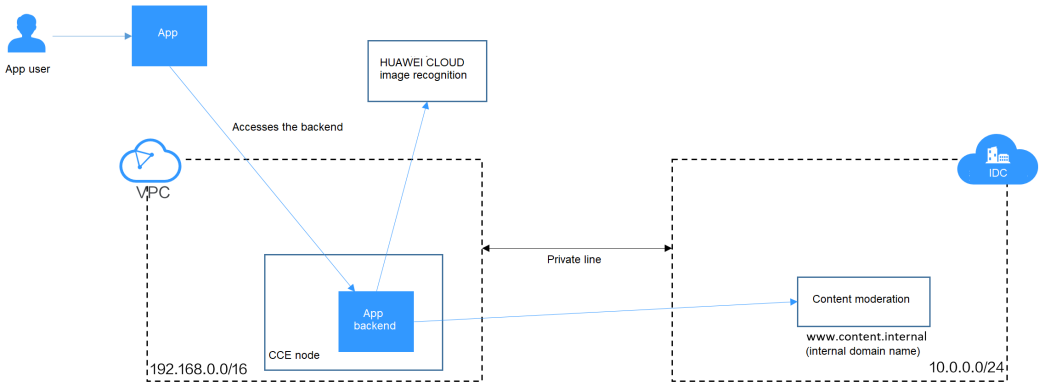
- The CCE cluster uses the Huawei Cloud DNS server, by default, to access the image recognition service.
- The CCE cluster uses the internal DNS server of the IDC to access the content moderation service deployed in the IDC.

In this case, the CCE cluster must be able to use both the Huawei Cloud DNS server and the internal DNS server of the IDC. If the DNS server on the CCE node points to the that of the IDC, the domain name of Huawei Cloud cannot be resolved. If the IP address of the IDC internal domain name is added to the **hosts** file, the configuration of the CCE node needs to be updated in real time when the IDC internal service IP changes. This is difficult to implement and may cause the CCE node to be unavailable.

NOTE

The content moderation and image recognition services are used only as examples.

Figure 10-17 Application deployment architecture



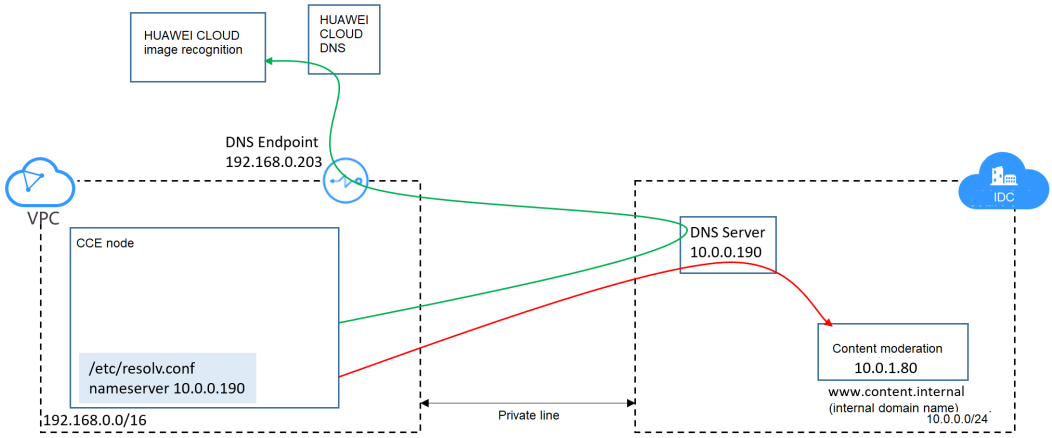
This section provides a solution for CCE clusters to resolve domain names of both on-premises IDCs and Huawei Cloud.

Solution 1: Using the DNS Endpoint for Cascading Resolution

You can use the VPC endpoint service to create a DNS endpoint cascaded with the IDC DNS server, so that nodes and containers in the CCE cluster can use the IDC DNS server for domain name resolution.

- If the Huawei Cloud domain name needs to be resolved, the request is forwarded to the DNS endpoint, and the Huawei Cloud DNS server is used to resolve the address and return the result.
- If the IDC domain name needs to be resolved, the IDC DNS server directly resolves the address and returns the result.

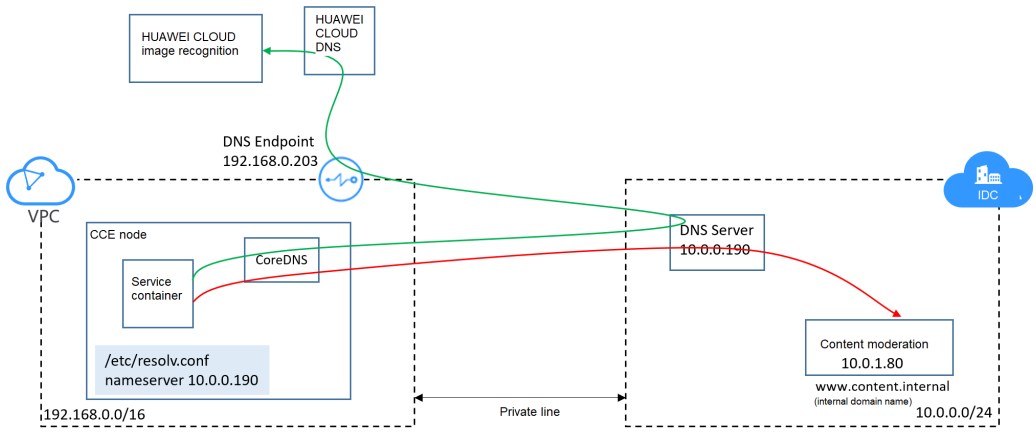
Figure 10-18 Accessing both the Huawei Cloud domain name and external domain name (for nodes)



For domain name resolution in a container, you can set the DNS policy to **ClusterFirst** when creating a pod. In this way, the domain name resolution requests of the container are directly sent to CoreDNS.

- If a cluster-internal domain name needs to be resolved, CoreDNS directly returns the resolution result.
- If an external domain name needs to be resolved, CoreDNS forwards the request to the IDC DNS server for resolution.

Figure 10-19 Accessing both the Huawei Cloud domain name and external domain name (for containers)

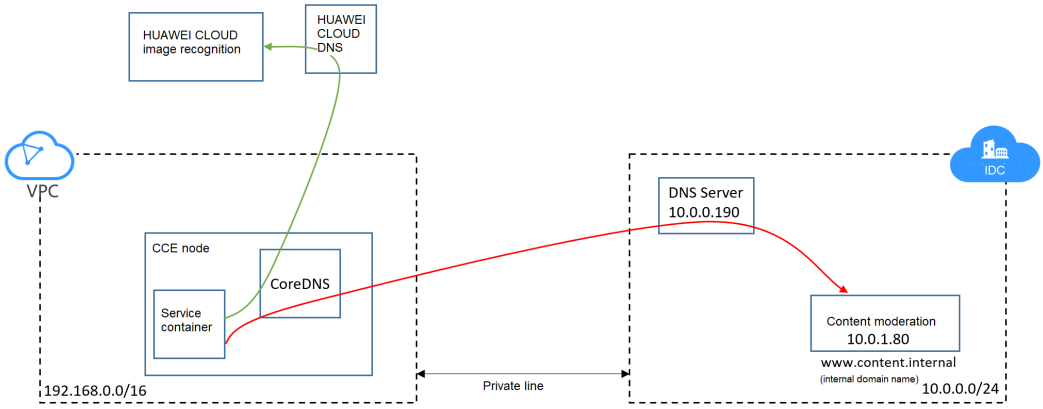


Solution 2: Changing the CoreDNS Configurations

Set the DNS policy to **ClusterFirst** when creating a pod so that the domain name resolution requests of containers are directly sent to CoreDNS.

- If a cluster-internal domain name needs to be resolved, CoreDNS directly returns the resolution result.
- If an external domain name needs to be resolved, CoreDNS forwards the request to the IDC DNS server for resolution.
- If a container accesses a Huawei Cloud internal domain name, the domain name is resolved by the internal DNS server of Huawei Cloud.

Figure 10-20 Domain name resolution in solution 2



Solution Comparison

Solution	Advantage	Disadvantage
Using the DNS endpoint for cascading resolution	External domain names can be resolved for containers and nodes in a CCE cluster.	An external DNS server is required to forward the requests for resolving internal domain names of Huawei Cloud, resulting in performance loss.
Changing the CoreDNS configuration	No external DNS server is required to forward the requests for resolving internal domain names of Huawei Cloud. Therefore, there is no performance loss.	• External domain names cannot be resolved on CCE cluster nodes. • The configuration will be lost if CoreDNS is upgraded or rolled back, and you need to reconfigure CoreDNS.

10.4.2 Solution 1: Using a DNS Endpoint for Cascading Resolution

Prerequisites

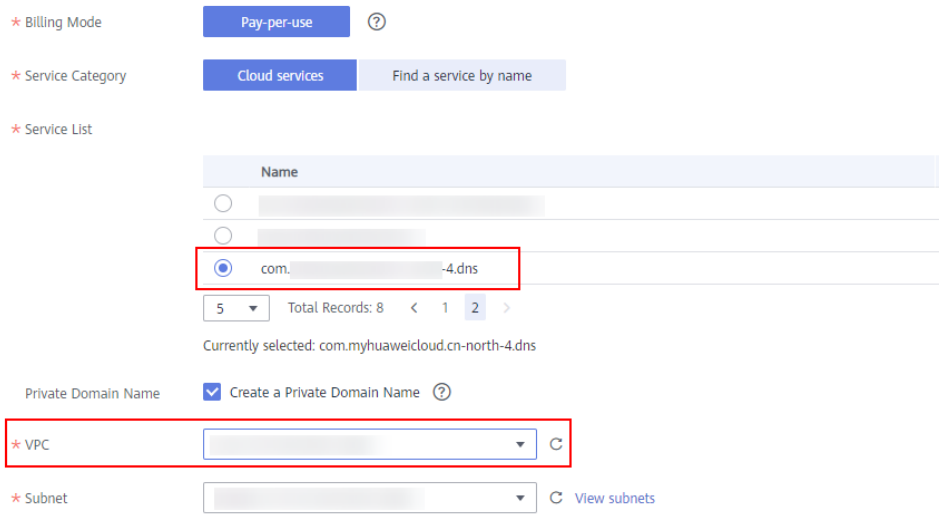
The VPC where the CCE cluster is deployed has been connected to the on-premises IDC through a private line (Direct Connect) or other channels. The IDC can access the IP addresses in the VPC CIDR block and CCE cluster container CIDR block. For details about how to create a Direct Connect connection, see **Getting Started with Direct Connect**.

Procedure

Step 1 Create a DNS endpoint in the VPC where the CCE cluster is deployed.

1. Access the **VPC Endpoint** page on the network console.
2. Click **Buy VPC Endpoint** in the upper right corner.
3. Select the DNS service and VPC where the CCE cluster is deployed.

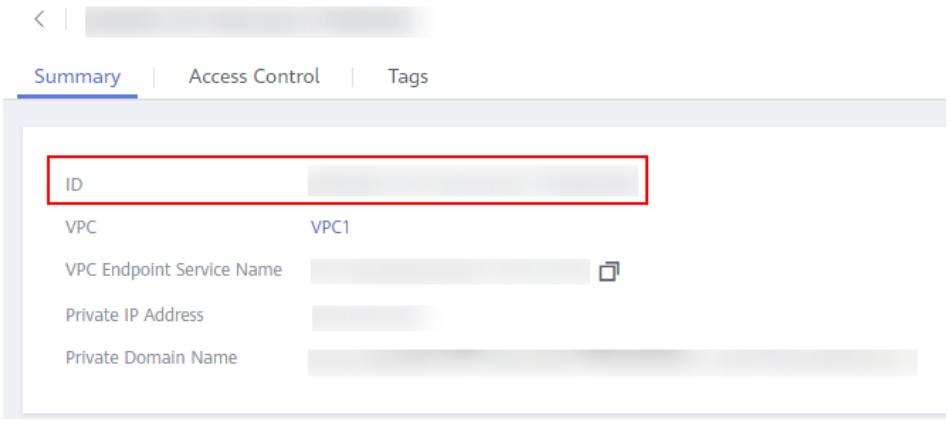
Figure 10-21 Creating a DNS endpoint



4. Click **Next** and then **Submit**.

   After the creation is complete, you can view the IP address of the DNS endpoint on its details page.

Figure 10-22 IP address of the DNS endpoint



Step 2 Configure cascading on the IDC DNS server.

NOTE

The configuration varies depending on the DNS server. The following configurations are used only as example.

BIND (a commonly used DNS server software) is used for the following demonstration.

In this step, you configure the DNS server to forward the requests of resolving Huawei Cloud internal domain names to the DNS endpoint created in the previous step.

For example, in BIND, you can add the lines marked in red to the **/etc/named.conf** file. **192.168.0.203** is the IP address of the DNS endpoint created in **Step 1**.

```
options {
        listen-on port 53 { any; };
        listen-on-v6 port 53 { ::1; };
        directory       "/var/named";
        dump-file       "/var/named/data/cache_dump.db";
        statistics-file "/var/named/data/named_stats.txt";
        memstatistics-file "/var/named/data/named_mem_stats.txt";
        recursing-file  "/var/named/data/named.recursing";
        secroots-file   "/var/named/data/named.secroots";
        allow-query     { any; };

        forward first;
        forwarders { 192.168.0.203; };

        .....
};
```

Step 3 Change the DNS configuration of the node in the CCE cluster.

You can use either of the following methods to change the settings.

Method 1:

Change the settings after the node is created.

1. Log in to the worker node of the CCE cluster.
2. In the **/etc/resolv.conf** file, change the value of nameserver to the IP address of the IDC DNS server.

```
# vi /etc/resolv.conf
nameserver 10.0.0.190
```


3. Run the following command to lock the **resolv.conf** file to prevent it from being automatically updated by Huawei Cloud.

   chattr +i /etc/resolv.conf

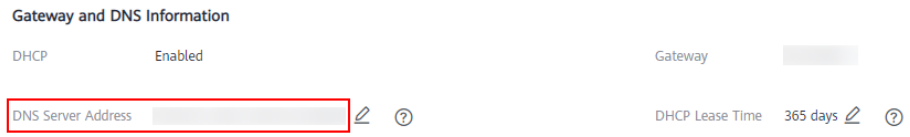
   For details about how to configure DNS, see **Configuring DNS**.

Method 2:

Change the DNS settings of the VPC subnet where the CCE cluster resides. In this way, the IP address of the specified DNS server is used in the **/etc/resolv.conf** file for newly created worker nodes.

Before using this method, ensure that the node can use the IDC DNS server to resolve the Huawei Cloud internal domain name. Otherwise, the node cannot be created. You are advised to commission the DNS server before you change the DNS settings of the VPC subnet.

Figure 10-23 Subnet DNS settings



Step 4 Configure the workload DNS policy.

When creating a workload, you can set **dnsPolicy** to **ClusterFirst** in the YAML file for domain name resolution in containers. This is also the default configuration in Kubernetes. For details about how to configure DNS for workloads, see **DNS Configuration**.

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
  - name: test
    image: nginx
  dnsPolicy: ClusterFirst
```

----End

Verification

After the configuration is complete, run the **dig** command on the cluster node. The command output shows that the IP address of the server is **10.0.0.190**, which indicates that the domain name is resolved by the IDC DNS server.

```
# dig cce.ap-southeast-1.myhuaweicloud.com

; <<>> DiG 9.9.4-61.1.h14.eulerosv2r7 <<>> cce.ap-southeast-1.myhuaweicloud.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24272
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
```



```
;cce.ap-southeast-1.myhuaweicloud.com. IN    A

;; ANSWER SECTION:
cce.ap-southeast-1.myhuaweicloud.com. 274 IN A       100.125.4.16

;; Query time: 4 msec
;; SERVER: 10.0.0.190#53(10.0.0.190)
;; WHEN: Tue Feb 23 19:16:08 CST 2021
;; MSG SIZE  rcvd: 76
```

Access a domain name of Huawei Cloud from the cluster node. The following output shows that the domain name is resolved to the corresponding IP address.

```
# ping cce.ap-southeast-1.myhuaweicloud.com
PING cce.ap-southeast-1.myhuaweicloud.com (100.125.4.16) 56(84) bytes of data.
```

Create a pod to access the Huawei Cloud domain name. The following output shows that the domain name can be resolved.

```
# kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --rm /bin/sh
If you do not see a command prompt, try pressing Enter.
# ping cce.ap-southeast-1.myhuaweicloud.com
PING cce.ap-southeast-1.myhuaweicloud.com (100.125.4.16) 56(84) bytes of data.
```

10.4.3 Solution 2: Changing the CoreDNS Configurations

Prerequisites

The VPC where the CCE cluster is deployed has been connected to the on-premises IDC through a private line (Direct Connect) or other channels. The IDC can access the IP addresses in the VPC CIDR block and CCE cluster container CIDR block. For details about how to create a Direct Connect connection, see **Getting Started with Direct Connect**.

Procedure

CoreDNS configurations are stored in the ConfigMap named **coredns**. You can find this ConfigMap in the kube-system namespace. Run the following command to view the default configurations.

kubectl get configmap coredns -n kube-system -oyaml

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: coredns
  namespace: kube-system
  selfLink: /api/v1/namespaces/kube-system/configmaps/coredns
  uid: d54ed5df-f4a0-48ec-9bc0-3efc1ac76af0
  resourceVersion: '21789515'
  creationTimestamp: '2021-03-02T09:21:55Z'
  labels:
    app: coredns
    k8s-app: coredns
    kubernetes.io/cluster-service: 'true'
    kubernetes.io/name: CoreDNS
    release: cceaddon-coredns
data:
  Corefile: |-
    .:5353 {
        bind {$POD_IP}
        cache 30
        errors
        health {$POD_IP}:8080
```



```
    kubernetes cluster.local in-addr.arpa ip6.arpa {
      pods insecure
      upstream /etc/resolv.conf
      fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf
    reload
}
```

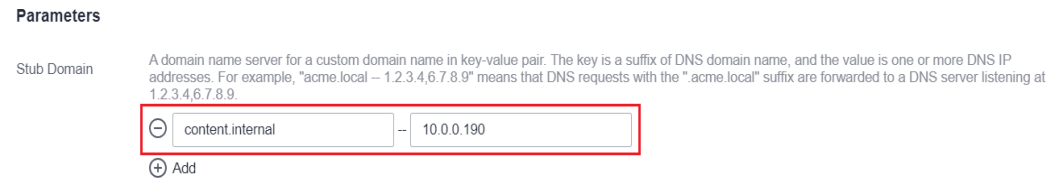
The preceding example shows that all CoreDNS configurations are defined in **Corefile**. By default, resolution requests of any domain name that does not belong to the Kubernetes cluster are directed to the DNS server specified by **forward**. In **forward . /etc/resolv.conf**, the first period (.) indicates all domain names, and **/etc/resolv.conf.** indicates the DNS server of the node.

If a specific external domain name needs to be resolved, you can add an extra configuration item. For example, if you want to forward the requests of resolving the domain name **content.internal** to the DNS server whose IP address is 10.0.0.190, perform the following operations to add configurations in **Corefile**.

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

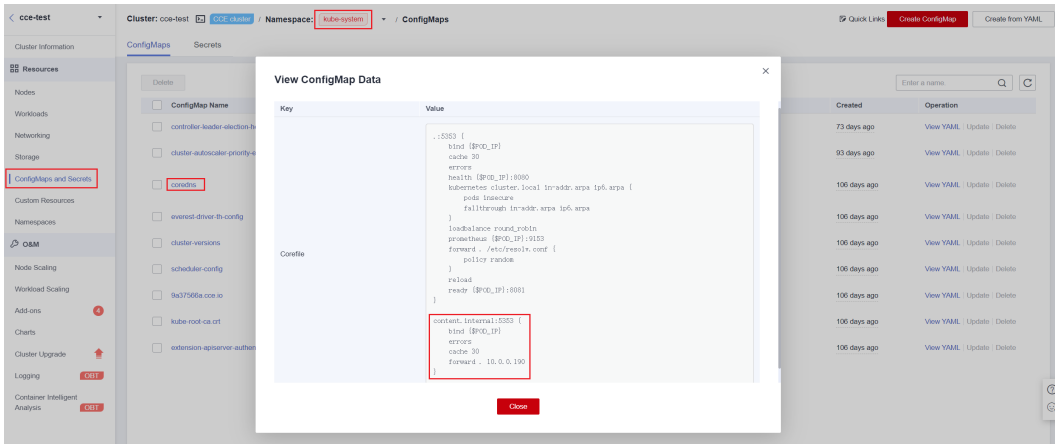
Step 2 In the navigation pane, choose **Add-ons**. On the displayed page, click **Edit** under **CoreDNS**.

Step 3 Add a stub domain in the **Parameters** area. A stub domain is a key-value pair. The key is a DNS suffix domain name, and the value is a DNS IP address or a group of DNS IP addresses. In this example, set the key-value pair to **content.internal --10.0.0.190**.



Step 4 Click **OK**.

Step 5 Choose **ConfigMaps and Secrets** in the navigation pane. In the **kube-system** namespace, view the coredns configuration data to check whether the update is successful.



Corresponding Corefile content:

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
content.internal:5353 {
    bind {$POD_IP}
    errors
    cache 30
    forward . 10.0.0.190
}
```

For details about other CoreDNS configurations, see **Customizing DNS Service**.

----End

Verification

Create a pod to access the IDC domain name. The following output shows that the domain name can be resolved.

```
web-terminal-568c6566df-c9jhl:~# kubectl run -i --tty --image tutum/dnsutils dnsutils --restart=Never --
rm /bin/sh
If you don't see a command prompt, try pressing enter.
# ping www.content.internal
PING www.content.internal (10.0.1.80) 56(84) bytes of data.
64 bytes from 10.0.1.80: icmp_seq=1 ttl=64 time=1.08 ms
64 bytes from 10.0.1.80: icmp_seq=2 ttl=64 time=0.337 ms
```

Access a Huawei Cloud domain name. The following output shows that the domain name can be resolved.

```
# ping cce.ap-southeast-1.myhuaweicloud.com
PING cce.ap-southeast-1.myhuaweicloud.com (100.125.4.16) 56(84) bytes of data.
```

Access the IDC domain name on the cluster node. The domain name cannot be pinged, indicating that the CoreDNS configurations do not affect the domain name resolution of the node.

10.5 Implementing Sticky Session Through Load Balancing

Concepts

Sticky sessions ensure continuity and consistency when you access applications. If a load balancer is deployed between a client and backend servers, connections may be forwarded to different servers for processing. Sticky sessions can resolve

this issue. After sticky session is enabled, requests from the same client will be continuously distributed to the same backend server through load balancing.

For example, in most online systems that require user identity authentication, a user needs to interact with the server for multiple times to complete a session. These interactions require continuity. If sticky session is not configured, the load balancer may allocate certain requests to different backend servers. Since user identity has not been authenticated on other backend servers, interaction exceptions such as a user login failure may occur.

Therefore, select a proper sticky session type based on the application environment.

Table 10-7 Sticky session types

OSI Layer	Listener Protocol and Networking	Sticky Session Type	Stickiness Duration	Scenarios Where Sticky Sessions Become Invalid
Layer 4	TCP- or UDP-compliant Services	**Source IP address**: The source IP address of each request is calculated using the consistent hashing algorithm to obtain a unique hashing key, and all backend servers are numbered. The system allocates the client to a particular server based on the generated key. This allows requests from the same IP address are forwarded to the same backend server.	• Default: 20 minutes • Maximum: 60 minutes • Range: 1 minute to 60 minutes	• Source IP addresses of the clients have changed. • Requests from the clients exceed the session stickiness duration.

OSI Layer	Listener Protocol and Networking	Sticky Session Type	Stickiness Duration	Scenarios Where Sticky Sessions Become Invalid
Layer 7	HTTP- or HTTPS-compliant ingresses	• **Load balancer cookie**: The load balancer generates a cookie after receiving a request from the client. All subsequent requests with the cookie will be routed to the same backend server. • **Application cookie**: The application deployed on the backend server generates a cookie after receiving the first request from the client. All subsequent requests with the same cookie will be routed to the same backend server.	• Default: 20 minutes • Maximum: 1440 minutes • Range: 1 minute to 1440 minutes	• If requests sent by the clients do not contain a cookie, sticky sessions will not take effect. • Requests from the clients exceed the session stickiness duration.

NOTE

When creating a load balancer, configure sticky sessions by setting **kubernetes.io/elb.lb-algorithm** to **ROUND_ROBIN** or **kubernetes.io/elb.lb-algorithm** to **LEAST_CONNECTIONS**. If you set **kubernetes.io/elb.lb-algorithm** is to **SOURCE_IP**, source IP address-based sticky sessions are supported. In this case, you do not need to configure sticky sessions again.

Layer 4 Sticky Sessions for Services

In Layer 4 mode, source IP address-based sticky sessions can be enabled, where hash routing is performed based on the client IP address.

Enabling Layer 4 Sticky Session in a CCE Standard Cluster

In a CCE standard cluster, to enable source IP address-based sticky session for a Service, ensure the following conditions are met:

1. **Service Affinity** of the Service must be set to **Node-level**, where the **externalTrafficPolicy** value of the Service must be **Local**.
2. Anti-affinity has been enabled on the backend applications of the Service to prevent all pods from being deployed on the same node.

Procedure

Step 1 Create an Nginx workload.

Set the number of pods to 3 and configure podAntiAffinity.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
      affinity:
        podAntiAffinity:                # Pod anti-affinity
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              topologyKey: kubernetes.io/hostname
```

Step 2 Create a LoadBalancer Service, for example, using an existing load balancer. The following shows an example YAML file for configuring source IP address-based sticky sessions:

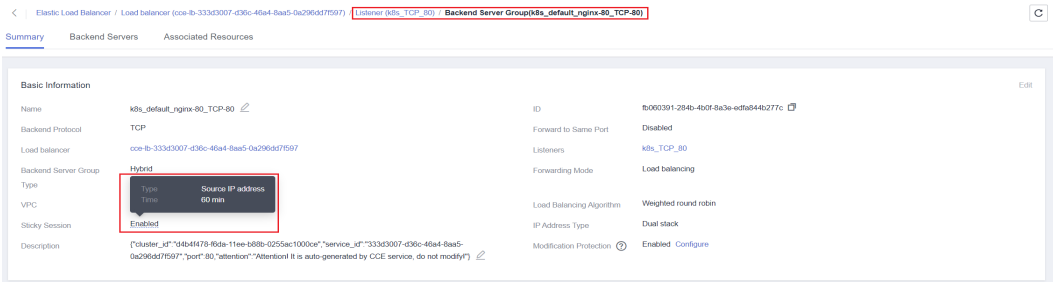
```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP    # Enable source IP address-based sticky session.
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local    # Node level Service affinity
  ports:
    - name: cce-service-0
```



```
    targetPort: 80
    nodePort: 32633
    port: 80
    protocol: TCP
type: LoadBalancer
```

Step 3 Log in to the ELB console and click the target load balancer. In the backend server group of the listener, check whether sticky session is enabled.

Figure 10-24 Enabled sticky session



----End

Enabling Layer 4 Sticky Session in a CCE Turbo Cluster

In a CCE Turbo cluster, enabling source IP address-based sticky session for a Service relies on the load balancer type.

- When a dedicated load balancer is used, passthrough networking is allowed between the load balancer and pods, and pods function as the backend server group of the load balancer. Therefore, you do not need to configure Service affinity or application anti-affinity when enabling source IP address-based sticky session for the Service.
- When a shared load balancer is used, to enable source IP address-based sticky session for a Service, ensure the following conditions are met:
  a. **Service Affinity** of the Service must be set to **Node-level**, where the **externalTrafficPolicy** value of the Service must be **Local**.
  b. Anti-affinity has been enabled on the backend applications of the Service to prevent all pods from being deployed on the same node.

Procedure

- **For dedicated load balancers**

  The following shows an example YAML file for configuring source IP address-based sticky sessions for a Service that uses an existing load balancer:

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP    # Enable source IP address-based sticky
session.
spec:
  selector:
    app: nginx
```



```
  externalTrafficPolicy: Local     # In CCE Turbo clusters, Service affinity does not need to be
configured if a dedicated load balancer is used.
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

- **For shared load balancers**

  a. Create an Nginx workload.

     Set the number of pods to 3 and configure podAntiAffinity.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
      affinity:
        podAntiAffinity:                  # Pod anti-affinity
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              topologyKey: kubernetes.io/hostname
```

  b. Create a LoadBalancer Service. The following shows an example YAML file for configuring source IP address-based sticky sessions for a Service that uses an existing load balancer:

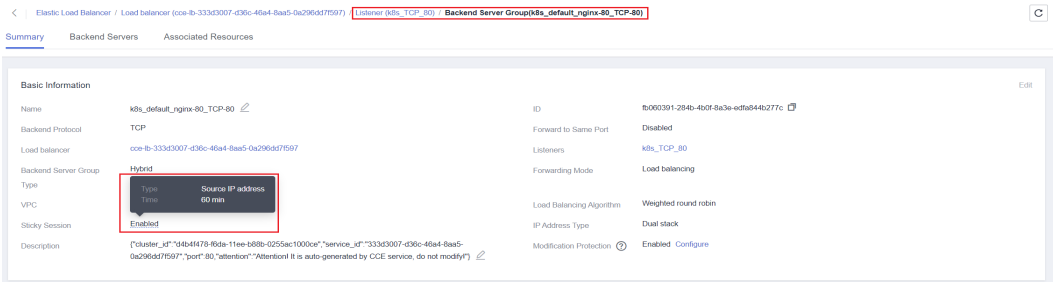
```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP     # Enable source IP address-based
sticky session.
spec:
  selector:
    app: nginx
```



```
externalTrafficPolicy: Local    # Node level Service affinity
ports:
  - name: cce-service-0
    targetPort: 80
    nodePort: 32633
    port: 80
    protocol: TCP
type: LoadBalancer
```

c. Log in to the ELB console and click the target load balancer. In the backend server group of the listener, check whether sticky session is enabled.

Figure 10-25 Enabled sticky session



Layer 7 Sticky Sessions for Ingresses

In Layer 7 mode, sticky sessions can be enabled using HTTP cookies or application cookies.

Enabling Layer 7 Sticky Session in a CCE Standard Cluster

To enable cookie-based sticky session on an ingress, ensure the following conditions are met:

1. **Service Affinity** of the ingress must be set to **Node-level**, where the **externalTrafficPolicy** value of the Service must be **Local**.
2. Anti-affinity must be enabled for the ingress workload to prevent all pods from being deployed on the same node.

Procedure

Step 1 Create an Nginx workload.

Set the number of pods to 3 and configure podAntiAffinity.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
```



```
        image: 'nginx:perl'
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
    imagePullSecrets:
      - name: default-secret
    affinity:
      podAntiAffinity:                  # Pod anti-affinity
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - nginx
            topologyKey: kubernetes.io/hostname
```

Step 2 Create a Service for the workload. This section uses a NodePort Service as an example.

Configure sticky sessions during the creation of a Service. An ingress can access multiple Services, and each Service can have different sticky sessions.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE      # HTTP cookie
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout":"1440"}'   # Session stickiness duration,
in minutes. The value ranges from 1 to 1440.
spec:
  selector:
    app: nginx
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 32633           # Custom node port
  type: NodePort
  externalTrafficPolicy: Local   # Node level Service affinity
```

You can also select **APP_COOKIE**.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: APP_COOKIE     # Select APP_COOKIE.
    kubernetes.io/elb.session-affinity-option: '{"app_cookie_name":"test"}'  # Application cookie name
...
```

Step 3 Create an ingress and associate it with the Service. The following uses an existing load balancer as an example. For details about how to automatically create a load balancer, see **Using kubectl to Create an ELB Ingress**.

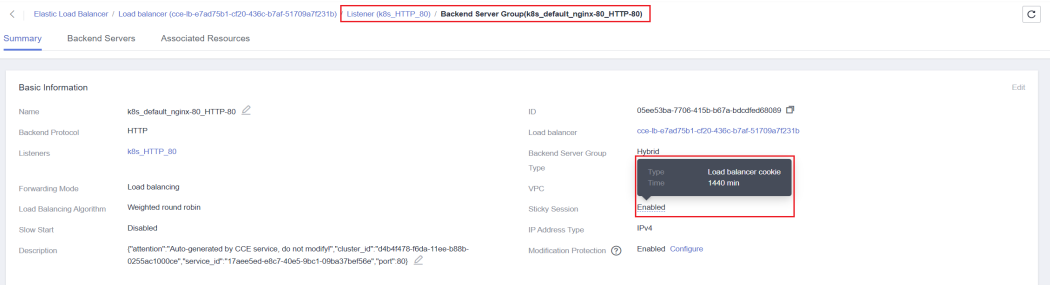
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
```



```
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.id: *****
spec:
  rules:
  - host: 'www.example.com'
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: nginx     # Service name
            port:
              number: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```

Step 4 Log in to the ELB console and click the target load balancer. In the backend server group of the listener, check whether sticky session is enabled.

Figure 10-26 Enabled sticky session



----End

Enabling Layer 7 Sticky Session in a CCE Turbo Cluster

Enable cookie-based sticky session on the ingress.

- When a dedicated load balancer is used, passthrough networking is allowed between the load balancer and pods, and pods function as the backend server group of the load balancer. Therefore, you do not need to configure Service affinity or application anti-affinity when enabling cookie-based sticky session for the ingress.
- When a shared load balancer is used, to enable cookie-based sticky session for an ingress, ensure the following conditions are met:
  a. **Service Affinity** of the ingress must be set to **Node-level**, where the **externalTrafficPolicy** value of the Service must be **Local**.
  b. Anti-affinity must be enabled for the ingress workload to prevent all pods from being deployed on the same node.

Procedure

- **For dedicated load balancers**

a. Create a Service for the workload. In a CCE Turbo cluster, the ingresses that use a dedicated load balancer must interconnect with ClusterIP Services.

Configure sticky sessions during the creation of a Service. An ingress can access multiple Services, and each Service can have different sticky sessions.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE      # HTTP cookie
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout":"1440"}'   # Session
stickiness duration, in minutes. The value ranges from 1 to 1440.
spec:
  selector:
    app: nginx
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 0
  type: ClusterIP
```

You can also select **APP_COOKIE**.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: APP_COOKIE     # Select APP_COOKIE.
    kubernetes.io/elb.session-affinity-option: '{"app_cookie_name":"test"}'  # Application cookie
name
...
```

b. Create an ingress and associate it with the Service. The following uses an existing load balancer as an example. For details about how to automatically create a load balancer, see **Using kubectl to Create an ELB Ingress**.

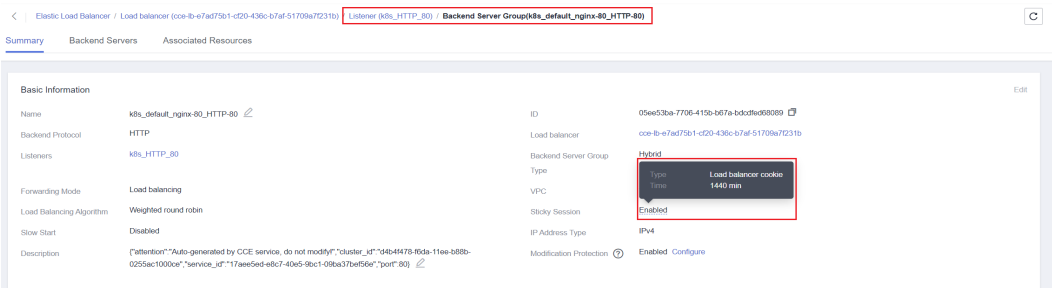
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.id: *****
spec:
  rules:
  - host: 'www.example.com'
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: nginx     # Service name
            port:
              number: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```



```
      pathType: ImplementationSpecific
ingressClassName: cce
```

c. Log in to the ELB console and click the target load balancer. In the backend server group of the listener, check whether sticky session is enabled.

Figure 10-27 Enabled sticky session



- **For shared load balancers**

a. Create an Nginx workload.

Set the number of pods to 3 and configure podAntiAffinity.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
      affinity:
        podAntiAffinity:                # Pod anti-affinity
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              topologyKey: kubernetes.io/hostname
```

b. Create a Service for the workload. In a CCE Turbo cluster, the ingresses that use a shared load balancer must interconnect with NodePort Services.

Configure sticky sessions during the creation of a Service. An ingress can access multiple Services, and each Service can have different sticky sessions.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE      # HTTP cookie
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout":"1440"}'   # Session
stickiness duration, in minutes. The value ranges from 1 to 1440.
spec:
  selector:
    app: nginx
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 32633            # Custom node port
  type: NodePort
  externalTrafficPolicy: Local   # Node level Service affinity
```

You can also select **APP_COOKIE**.

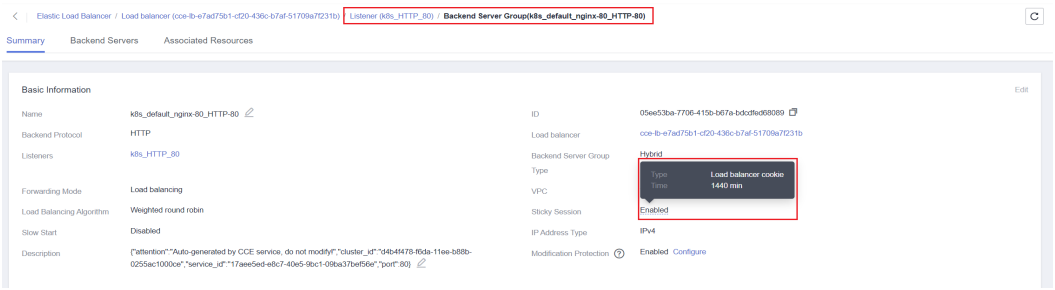
```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN      # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: APP_COOKIE      # Select APP_COOKIE.
    kubernetes.io/elb.session-affinity-option: '{"app_cookie_name":"test"}'  # Application cookie
name
...
```

c. Create an ingress and associate it with the Service. The following uses an existing load balancer as an example. For details about how to automatically create a load balancer, see **Using kubectl to Create an ELB Ingress**.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.id: *****
spec:
  rules:
  - host: 'www.example.com'
    http:
      paths:
      - path: '/'
        backend:
          service:
            name: nginx     # Service name
            port:
              number: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
  ingressClassName: cce
```


d. Log in to the ELB console and click the target load balancer. In the backend server group of the listener, check whether sticky session is enabled.

Figure 10-28 Enabled sticky session

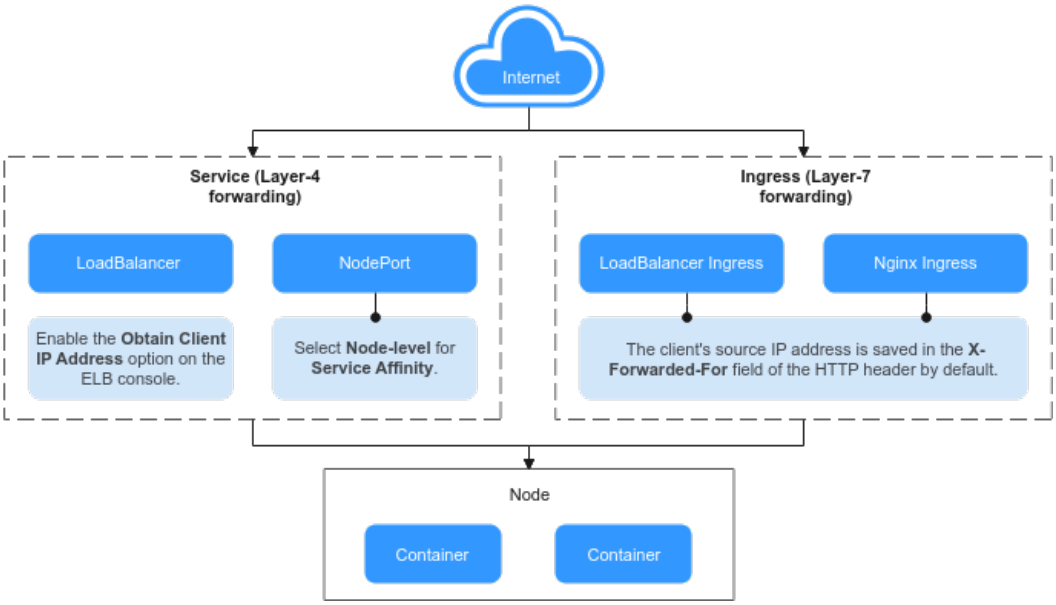


10.6 Obtaining the Client Source IP Address for a Container

In containers, multiple types of proxy servers may exist between a client and the container servers. After an external request is forwarded for multiple times, the source IP address of the client cannot be transmitted to the containers. As a result, Services in the containers cannot obtain the real source IP addresses of the client.

Description

Figure 10-29 Obtaining the source IP addresses from the containers



Layer-7 forwarding:

Ingresses: If this access mode is used, the client's source IP address is saved in the **X-Forwarded-For** field of the HTTP header by default. No other configuration is required.

- Huawei LoadBalancer Ingresses use ELB for Layer 7 network access between the Internet and internal network (in the same VPC) based on the ELB service.
- The Nginx Ingresses implement Layer 7 network access based on nginx-ingress. The backend Service type can be either **ClusterIP** or **NodePort**.

Layer-4 forwarding:

- LoadBalancer: Use ELB to achieve load balancing. You can manually enable the **Transfer Client IP Address** option for TCP and UDP listeners of shared load balancers. By default, the **Transfer Client IP Address** option is enabled for TCP and UDP listeners of dedicated load balancers. You do not need to manually enable it.
- NodePort: The container port is mapped to the node port. If the cluster-level affinity is selected, access requests will be forwarded through the node and the client source IP address cannot be obtained. If the node-level affinity is selected, access requests will not be forwarded and the client source IP address can be obtained.

NOTE

If Istio is used, you can obtain the source IP address by referring to **How Do I Obtain the Actual Source IP Address of a Client After a Service Is Added into Istio?**

Scenarios in Which Source IP Address Can Be Obtained

Due to network model differences, CCE does not allow obtaining source IP addresses in some scenarios, as listed in **Table 10-8**. "-" in the table indicates that this scenario does not exist.

Table 10-8 Scenarios in which source IP addresses can be obtained

Level-1 Category	Level-2 Category	Load Balancer Type	VPC and Container Tunnel Network Models	Cloud Native Network 2.0 Model (CCE Turbo Clusters)	Reference
Layer-7 forwarding (ingress)	ELB	Shared	Supported	Supported	**ELB Ingress**
		Dedicated	Supported	Supported	
	Nginx (interconnected with the nginx-ingress add-on)	Shared	Supported	Not supported	**Nginx Ingress**
		Dedicated	Supported	Supported	

Level-1 Category	Level-2 Category	Load Balancer Type	VPC and Container Tunnel Network Models	Cloud Native Network 2.0 Model (CCE Turbo Clusters)	Reference
Layer-4 forwarding (Service)	LoadBalancer	Shared	Supported	Not supported (supported by workloads that use hostNetwork)	LoadBalancer
		Dedicated	Supported	Supported	
	NodePort	-	Supported	Not supported (supported by workloads that use hostNetwork)	NodePort

ELB Ingress

For the ELB Ingresses (using HTTP- or HTTPS-compliant), the function of obtaining the source IP addresses of the client is enabled by default. No other operation is required.

The real IP address is placed in the **X-Forwarded-For** HTTP header field by the load balancer in the following format:

```
X-Forwarded-For: IP address of the client,Proxy server 1-IP address,Proxy server 2-IP address,...
```

If you use this method, the first IP address obtained is the IP address of the client.

Nginx Ingress

For the Nginx Ingresses, perform the following operations.

NOTE

In the Cloud Native Network 2.0 model, source IP addresses cannot be obtained if a shared load balancer is used when an Ingress is interconnected with the nginx-ingress add-on. For details, see **Scenarios in Which Source IP Address Can Be Obtained**. To obtain the source IP, uninstall the nginx-ingress add-on and use a dedicated load balancer during reinstallation.

Step 1 Take the Nginx workload as an example. Before configuring the source IP address, obtain the access logs. **nginx-c99fd67bb-ghv4q** indicates the pod name.


```
kubectl logs nginx-c99fd67bb-ghv4q
```

Information similar to the following is displayed:

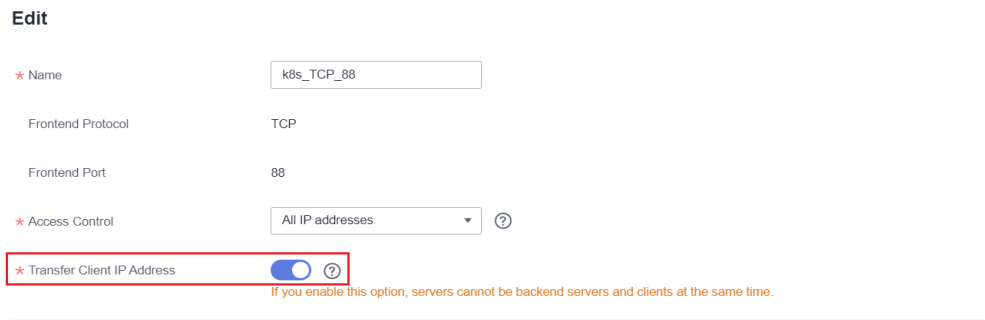
```
...
10.0.0.7 - - [17/Aug/2023:01:30:11 +0000] "GET / HTTP/1.1" 200 19 "http://114.114.114.114:9421/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 Edg/115.0.1901.203" "100.125.**.**"
```

100.125..**** specifies the CIDR block of the load balancer, indicating that the traffic is forwarded through the load balancer.

Step 2 Go to the ELB console and enable the function of obtaining the client IP address of the listener corresponding to the load balancer. **Transparent transmission of source IP addresses is enabled for dedicated load balancers by default. You do not need to manually enable this function.**

1. Log in to the ELB console.
2. Click  in the upper left corner of the management console and select a region and a project.
3. Click **Service List**. Under **Networking**, click **Elastic Load Balance**.
4. On the **Load Balancers** page, click the name of the load balancer.
5. Click the **Listeners** tab, locate the row containing the target listener, and click **Edit**. If modification protection exists, disable the protection on the basic information page of the listener and try again.
6. Enable **Transfer Client IP Address**.

Figure 10-30 Enabling the function



Step 3 Edit the nginx-ingress add-on. In the nginx configuration parameter area, configure the configuration fields and information. For details about the parameter range, see **community document**. After the configuration is complete, update the add-on.

```
{
  "enable-real-ip": "true",
  "forwarded-for-header": "X-Forwarded-For",
  "proxy-real-ip-cidr": "100.125.0.0/16",
  "keep-alive-requests": "100"
}
```


NOTE

The **proxy-real-ip-cidr** parameter indicates the CIDR block of the proxy server.

- For shared load balancers, add CIDR block 100.125.0.0/16 (reserved only for load balancers and therefore, there is no risk) and the high-defense CIDR block.
- For dedicated load balancers, add the CIDR block of the VPC subnet where the ELB resides.

For details, see **How Can I Transfer the IP Address of a Client?**

Step 4 Access the workload again and view the new access log.

```
...
10.0.0.7 - - [17/Aug/2023:02:43:11 +0000] "GET / HTTP/1.1" 304 0 "http://114.114.114.114:9421/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/115.0.0.0 Safari/537.36 Edg/115.0.1901.203" "124.**.**.**"
```

The source IP address of the client is obtained.

----End

LoadBalancer

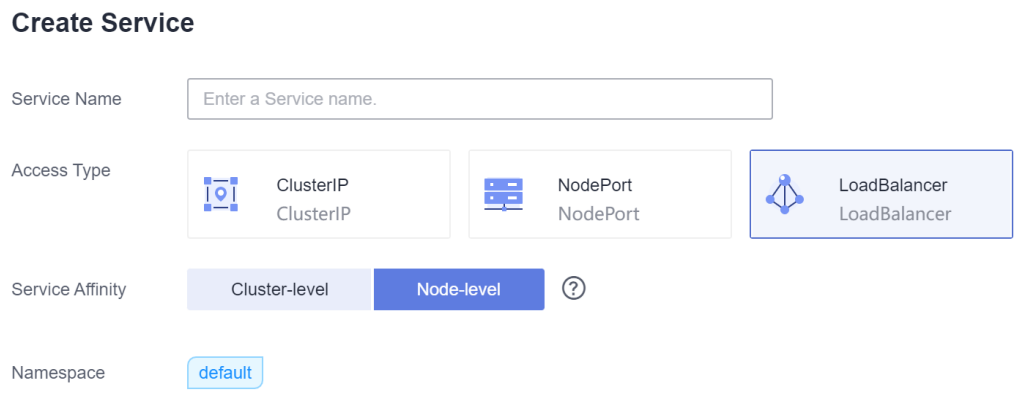
For a LoadBalancer Service, different types of clusters obtain source IP addresses in different scenarios. In some scenarios, source IP addresses cannot be obtained currently. For details, see **Scenarios in Which Source IP Address Can Be Obtained**.

- CCE Clusters (using VPC or Tunnel network): Source IP addresses can be obtained when either a shared or dedicated load balancer is used.
- CCE Turbo Clusters (using the Cloud Native Network 2.0): Source IP addresses can be obtained for dedicated load balancers, and for shared load balancers with hostNetwork enabled.

VPC and Container Tunnel Network Models

To obtain source IP addresses, perform the following steps:

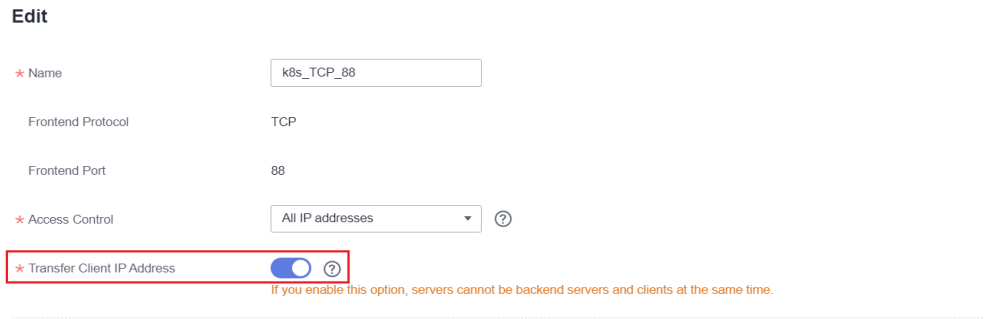
Step 1 When creating a LoadBalancer Service on the CCE console, set **Service Affinity** to **Node-level** instead of **Cluster-level**.



Step 2 Go to the ELB console and enable the function of obtaining the client IP address of the listener corresponding to the load balancer. **Transparent transmission of source IP addresses is enabled for dedicated load balancers by default. You do not need to manually enable this function.**

1. Log in to the ELB console.
2. Click in the upper left corner of the management console and select a region and a project.
3. Click **Service List**. Under **Networking**, click **Elastic Load Balance**.
4. On the **Load Balancers** page, click the name of the load balancer.
5. Click the **Listeners** tab, locate the row containing the target listener, and click **Edit**. If modification protection exists, disable the protection on the basic information page of the listener and try again.
6. Enable **Transfer Client IP Address**.

Figure 10-31 Enabling the function

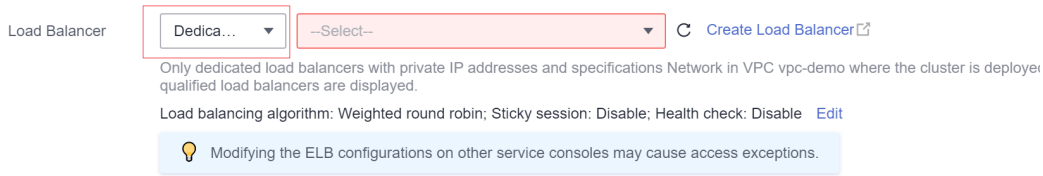


----End

Cloud Native Network 2.0 Model (CCE Turbo Clusters)

In the Cloud Native Network 2.0 model, when a shared load balancer is used for load balancing, the service affinity cannot be set to **Node-level**. As a result, source IP addresses cannot be obtained. To obtain a source IP address, you must use a **dedicated load balancer**. External access to the container does not need to pass through the forwarding plane.

By default, transparent transmission of source IP addresses is enabled for dedicated load balancers. You do not need to manually enable **Transfer Client IP Address** on the ELB console. Instead, you only need to select a dedicated load balancer when creating an ENI LoadBalancer Service on the CCE console.



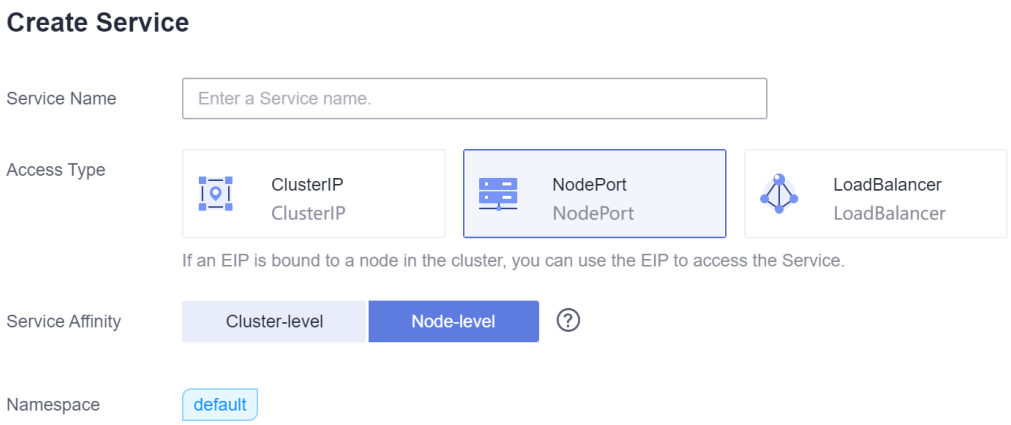
NodePort

Set the service affinity of a NodePort Service to **Node-level** instead of **Cluster-level**. That is, set **spec.externalTrafficPolicy** of the Service to **Local**.

NOTE

When a node (using Cloud Native Network 2.0) accesses a NodePort Service, source IP addresses can be obtained only when hostNetwork is enabled for workloads.

Figure 10-32 Selecting a node-level affinity



10.7 Increasing the Listening Queue Length by Configuring Container Kernel Parameters

Application Scenarios

net.core.somaxconn indicates the maximum number of half-open connections that can be backlogged in a listening queue. The default value is 128. If the queue is overloaded, increase the listening queue length.

Procedure

Step 1 Modify kubelet configurations.

You can use either of the following methods to modify the kubelet parameters:

- **Modifying kubelet parameters in the node pool (only for clusters of v1.15 or later)**

  Log in to the CCE console, go to the cluster details page, choose **More** > **Manage** in the node pool, and modify the kubelet parameters.

  Figure 10-33 Node pool configuration

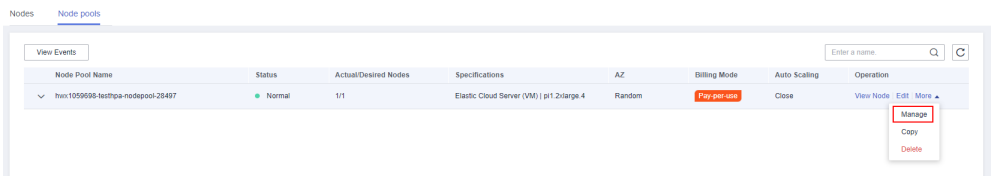
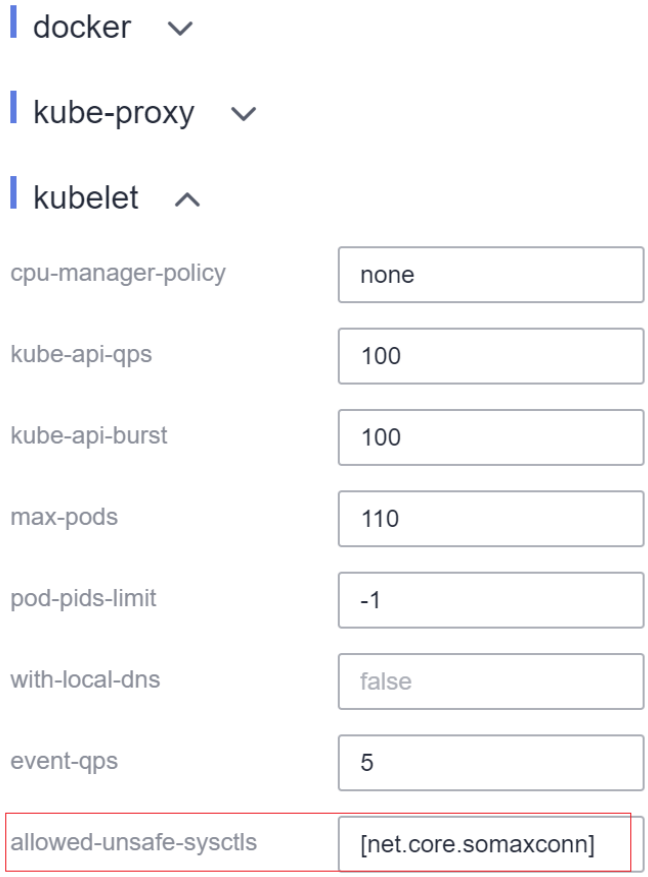


Figure 10-34 Modifying kubelet parameters



- **Modifying kubelet parameters of the node**
  a. Log in to the node.
  b. Edit the **/opt/cloud/cce/kubernetes/kubelet/kubelet** file. In versions earlier than 1.15, the file is **/var/paas/kubernetes/kubelet/kubelet**.
     Enable net.core.somaxconn.

     ```
     --allowed-unsafe-sysctls=net.core.somaxconn
     ```

     ```
     [root@test-565556-38106 ~]# cat /var/paas/kubernetes/kubelet/kubelet
     DAEMON_ARGS=" --bootstrap-kubeconfig=/var/paas/kubernetes/kubelet/boot.conf --cert-dir=/var/paas/kubernetes/kubelet/pki --rotate
     -certificates=true --network-plugin=cni --cni-conf-dir=/etc/cni/net.d/ --node-ip=192.168.116.149 --provider-id=acb56430-9a20-11e
     9-bf1e-0255ac101f9c --kubeconfig=/var/paas/kubernetes/kubelet/kubeconfig --config=/var/paas/kubernetes/kubelet/kubelet_config.ya
     ml --authentication-token-webhook=true --pod-infra-container-image=cce-pause:2.0 --root-dir=/mnt/paas/kubernetes/kubelet --hostn
     ame-override=192.168.116.149 --allow-privileged=True --v=2 --node-labels="os.name=EulerOS_2.0_SP5,os.version=3.10.0-862.14.0.1.h
     147.eulerosv2r7.x86_64,os.architecture=amd64,failure-domain.beta.kubernetes.io/zone=cn-north-1a,failure-domain.beta.kubernetes.i
     o/region=cn-north-1,kubernetes.io/availablezone=cn-north-1a,failure-domain.beta.kubernetes.io/is-baremetal='false'" --machine_id
     _file=/var/paas/conf/server.conf --cadvisor-port=4194 --allowed-unsafe-sysctls=net.core.somaxconn"
     [root@test-565556-38106 ~]#
     ```

  c. Restart kubelet.
     systemctl restart kubelet
     Check the kubelet status.
     systemctl status kubelet

  NOTE

  After the kubelet configurations are changed for a cluster of v1.13 or earlier, the configurations will be restored if the cluster is upgraded to a later version.

Step 2 Create a pod security policy.

Starting from **v1.17.17**, CCE enables pod security policies for kube-apiserver. Add **net.core.somaxconn** to **allowedUnsafeSysctls** of a pod security policy to make

the policy take effect. (This configuration is not required for clusters earlier than v1.17.17.)

- For details about CCE security policies, see **Pod Security Policies**.
- For details about Kubernetes security policies, see **PodSecurityPolicy**.
- For clusters with **net.core.somaxconn** enabled, add this configuration to **allowedUnsafeSysctls** of the corresponding pod security policy. For example, create a pod security policy as follows:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
  - net.core.somaxconn
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  fsGroup:
    rule: RunAsAny
  hostIPC: true
  hostNetwork: true
  hostPID: true
  hostPorts:
  - max: 65535
    min: 0
  privileged: true
  runAsGroup:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  volumes:
  - '*'
```

After creating the pod security policy **sysctl-psp**, configure RBAC permission control for it.

An example is as follows:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: sysctl-psp
rules:
  - apiGroups:
      - "*"
    resources:
      - podsecuritypolicies
    resourceNames:
      - sysctl-psp
    verbs:
      - use

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: sysctl-psp
roleRef:
  kind: ClusterRole
  name: sysctl-psp
  apiGroup: rbac.authorization.k8s.io
```



```
subjects:
- kind: Group
  name: system:authenticated
  apiGroup: rbac.authorization.k8s.io
```

Step 3 Create a workload, set kernel parameters, and configure the affinity with the node in **Step 1**.

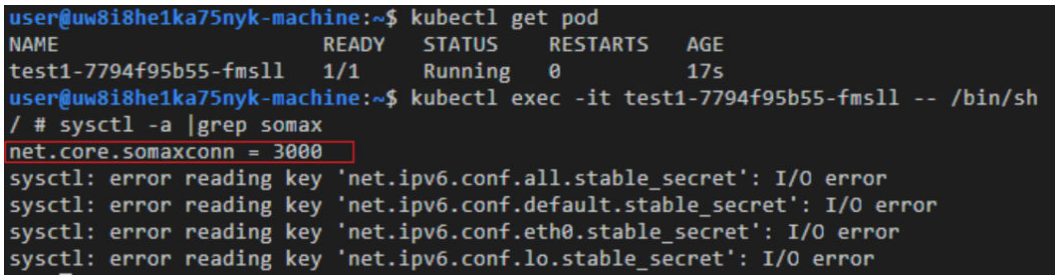
```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    description: ''
  labels:
    appgroup: ''
  name: test1
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test1
  template:
    metadata:
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
      labels:
        app: test1
    spec:
      containers:
        - image: 'nginx:1.14-alpine-perl'
          name: container-0
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
            limits:
              cpu: 250m
              memory: 512Mi
      imagePullSecrets:
        - name: default-secret
      securityContext:
        sysctls:
          - name: net.core.somaxconn
            value: '3000'
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: kubernetes.io/hostname
                    operator: In
                    values:
                      - 192.168.x.x       # Node name.
```

Step 4 Log in to the node where the workload is deployed, access the container, and check whether the parameter configuration takes effect.

Run the following command in the container to check whether the configuration takes effect:

sysctl -a |grep somax

Figure 10-35 Viewing the parameter configuration



----End

10.8 Enabling Passthrough Networking for LoadBalancer Services

Background

A Kubernetes cluster can publish applications running on a group of pods as Services, which provide unified layer-4 access entries. For a Loadbalancer Service, kube-proxy configures the LoadbalanceIP in **status** of the Service to the local forwarding rule of the node by default. When a pod accesses the load balancer from within the cluster, the traffic is forwarded within the cluster instead of being forwarded by the load balancer.

kube-proxy is responsible for intra-cluster forwarding. kube-proxy has two forwarding modes: iptables and IPVS. iptables is a simple polling forwarding mode. IPVS has multiple forwarding modes but it requires modifying the startup parameters of kube-proxy. Compared with iptables and IPVS, load balancers provide more flexible forwarding policies as well as health check capabilities.

Solution

CCE supports passthrough networking. You can configure the **annotation** of **kubernetes.io/elb.pass-through** for the Loadbalancer Service. Intra-cluster access to the Service load balancer address is then forwarded to backend pods by the load balancer.

- CCE clusters

  When a LoadBalancer Service is accessed within the cluster, the access is forwarded to the backend pods using iptables/IPVS by default.

  When a LoadBalancer Service (configured with elb.pass-through) is accessed within the cluster, the access is first forwarded to the load balancer, then the nodes, and finally to the backend pods using iptables/IPVS.

- CCE Turbo clusters

  When a client accesses a LoadBalancer Service from within the cluster, pass-through is used by default. In this case, the client directly accesses the load balancer private network IP address and then access a container through the load balancer.

Constraints

- After passthrough networking is configured for a dedicated load balancer, in a CCE standard cluster, pods that run on the same node as the workload and pods that run on the same node cannot be accessed through the LoadBalancer Service.
- Passthrough networking is not supported for clusters of v1.15 or earlier.
- In IPVS network mode, the pass-through settings of Service connected to the same ELB must be the same.
- If node-level (local) service affinity is used, **kubernetes.io/elb.pass-through** is automatically set to **onlyLocal** to enable pass-through.

Procedure

This section describes how to create a Deployment using an Nginx image and create a Service with passthrough networking enabled.

Step 1 Use the Nginx image to create a Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

Step 2 For a LoadBalance Service type, set **kubernetes.io/elb.pass-through** to **true**. In this example, a shared load balancer named **james** is automatically created.

For details about how to create a LoadBalancer Service, see **LoadBalancer**.

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.pass-through: "true"
    kubernetes.io/elb.class: union
    kubernetes.io/elb.autocreate: '{"type":"public","bandwidth_name":"cce-
bandwidth","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_ty
pe":"5_bgp","name":"james"}'
  labels:
    app: nginx
  name: nginx
spec:
  externalTrafficPolicy: Local
```

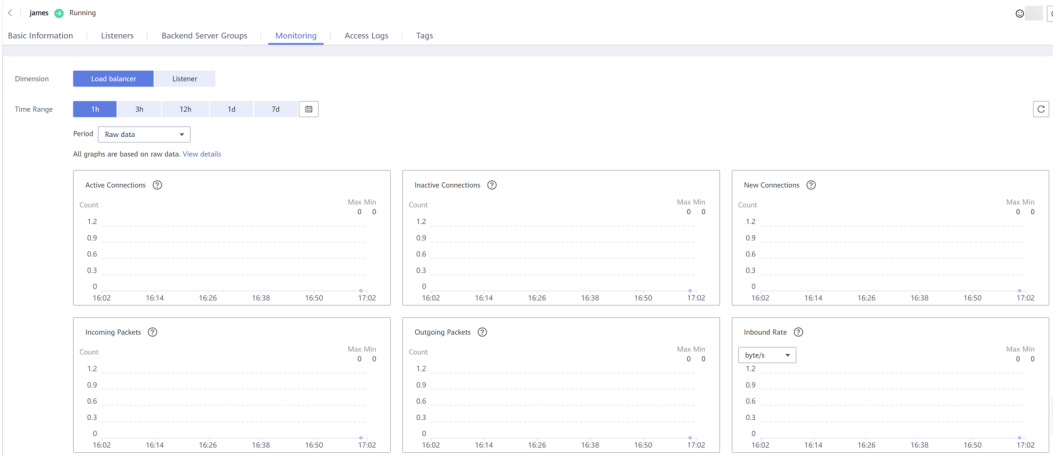


```
ports:
- name: service0
  port: 80
  protocol: TCP
  targetPort: 80
selector:
  app: nginx
type: LoadBalancer
```

----End

Verification

Check the ELB load balancer corresponding to the created Service. The load balancer name is **james**. The number of ELB connections is **0**, as shown in the following figure.



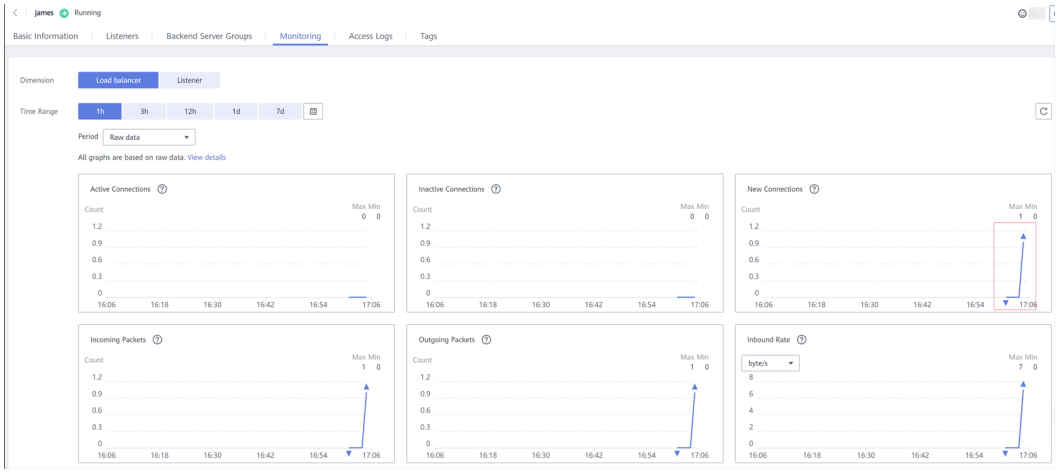
Use kubectl to connect to the cluster, go to an Nginx container, and access the ELB address. The access is successful.

```
# kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
nginx-7c4c5cc6b5-vpncx   1/1     Running   0          9m47s
nginx-7c4c5cc6b5-xj5wl   1/1     Running   0          9m47s
# kubectl exec -it nginx-7c4c5cc6b5-vpncx -- /bin/sh
# curl 120.46.141.192
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
```



```
</body>
</html>
```

Wait for a period of time and view the ELB monitoring data. A new access connection is created for the ELB, indicating that the access passes through the ELB load balancer as expected.



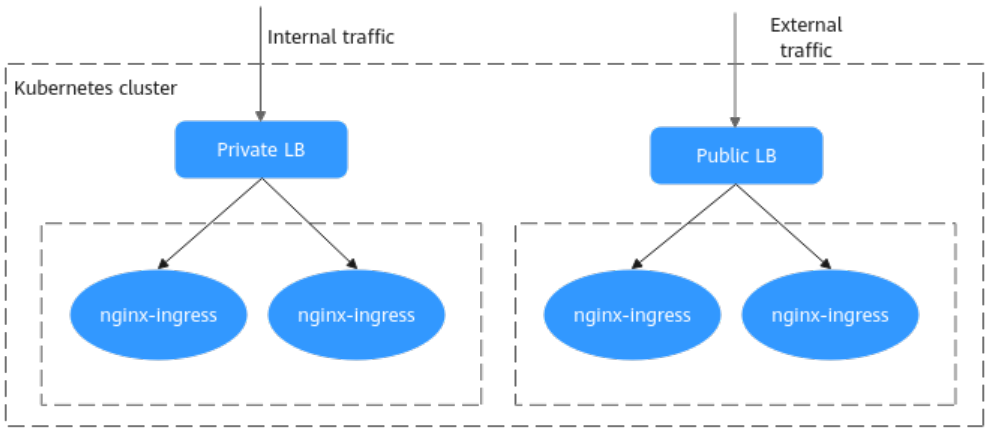
10.9 Deploying Nginx Ingress Controllers Using a Chart

10.9.1 Deploying NGINX Ingress Controller in Custom Mode

Background

Nginx Ingress Controller is a popular open source ingress controller in the industry and is widely used. Large-scale clusters require multiple ingress controllers to distinguish different traffic. For example, if some services in a cluster need to be accessed through a public network ingress, but some internal services cannot be accessed through a public network and can only be accessed by other services in the same VPC, you can deploy two independent Nginx Ingress Controllers and bind two different load balancers.

Figure 10-36 Application scenario of multiple Nginx ingresses



Solution

You can use either of the following solutions to deploy multiple NGINX ingress controllers in the same cluster.

- (Recommended) Install the NGINX Ingress Controller add-on and deploy multiple instances in the same cluster with one click. For details, see **Nginx Ingress Controller**.

  For clusters 1.23, the add-on of 2.2.52 or later must be installed. For clusters 1.23 or later, the add-on of 2.5.4 or later must be installed.

- Install the open source Helm package. The parameters to be configured in this solution are complex. You need to configure the **ingress-class** parameter (default value: **nginx**) to declare the listening ranges of different NGINX ingress controllers. In this way, when creating an ingress, you can select different NGINX ingress controllers to distinguish traffic.

Prerequisites

- Public images may need to be pulled during the installation. Therefore, bind an EIP to the node.

Constraints

- If multiple Nginx Ingress Controllers are deployed, each Controller needs to interconnect with a load balancer. Ensure that the load balancer has at least two listeners and ports 80 and 443 are not occupied by listeners. If dedicated load balancers are used, specify the network type.
- When the nginx-ingress template and image provided by the community are used, CCE does not provide additional maintenance for service loss caused by community software defects. **Exercise caution when serving commercial purposes.**

Deploying an Nginx Ingress Controller

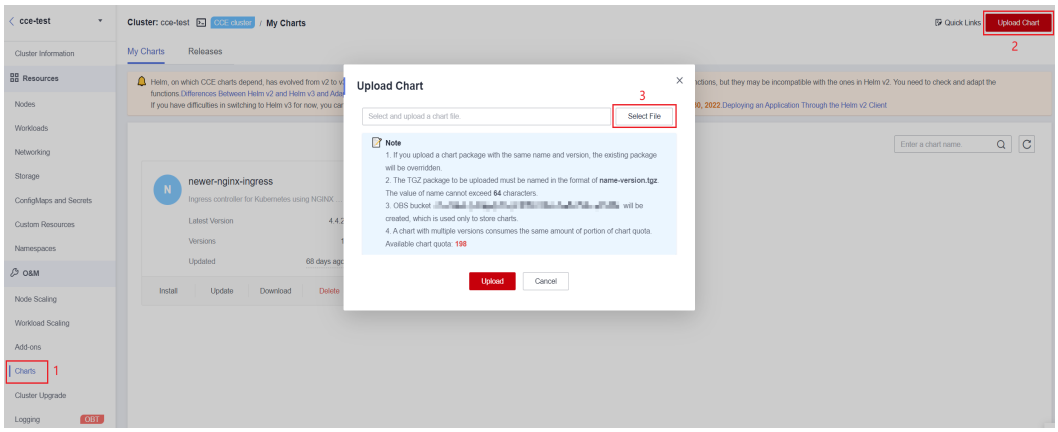
You can perform the following steps to deploy another independent NGINX Ingress Controller in the cluster.

Step 1 Obtain a chart.

Go to the **chart page**, select a proper version, and download the Helm chart in **.tgz** format. This section uses the chart of version **4.4.2** as an example. This chart applies to CCE clusters of v1.21 or later. The configuration items in the chart may vary according to the version. The configuration in this section takes effect only for the chart of **4.4.2** version.

Step 2 Upload the chart.

1. Log in to the CCE console and click the cluster name to access the cluster console. In the navigation pane, choose **App Templates** and click **Upload Chart** in the upper right corner.
2. Click **Select File**, select the chart to be uploaded, and click **Upload**.



Step 3 Customize the value.yaml file.

You can create a **value.yaml** configuration file on the local PC to configure workload installation parameters. During workload installation, you only need to import this configuration file for customized installation. Other unspecified parameters will use the default settings.

The configuration content is as follows:

```
controller:
  image:
    repository: registry.k8s.io/ingress-nginx/controller
    registry: ""
    image: ""
    tag: "v1.5.1"  # Controller version
    digest: ""
  ingressClassResource:
    name: ccedemo        # The name of each Ingress Controller in the same cluster must be unique and cannot be nginx or cce.
    controllerValue: "k8s.io/ingress-nginx-demo"  # The listening identifier of each Ingress Controller in the same cluster must be unique and cannot be set to k8s.io/ingress-nginx.
  ingressClass: ccedemo   # The name of each Ingress Controller in the same cluster must be unique and cannot be nginx or cce.
  service:
    annotations:
      kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0     #ELB ID
      kubernetes.io/elb.class: performance  # This annotation is required only for dedicated load balancers.
  config:
    keep-alive-requests: 100
  extraVolumeMounts: # Mount the /etc/localtime file on the node to synchronize the time zone.
    - name: localtime
      mountPath: /etc/localtime
      readOnly: true
  extraVolumes:
    - name: localtime
      type: Hostpath
      hostPath:
        path: /etc/localtime
  admissionWebhooks: # Disable webhook authentication.
    enabled: false
    patch:
      enabled: false
  resources: # Set the controller's resource limit, which can be customized.
    requests:
      cpu: 200m
      memory: 200Mi
defaultBackend: # Set defaultBackend.
  enabled: true
  image:
    repository: registry.k8s.io/defaultbackend-amd64
    registry: ""
    image: ""
```



```
tag: "1.5"
digest: ""
```

For details about the preceding parameters, see **Table 10-9**.

Step 4 Create a release.

1. Log in to the CCE console and click the cluster name to access the cluster console. In the navigation pane, choose **App Templates**.
2. In the list of uploaded charts, click **Install**.
3. Set **Release Name**, **Namespace**, and **Version**.
4. Click **Select File** next to **Configuration File**, select the YAML file created locally, and click **Install**.
5. On the **Releases** tab page, you can view the installation status of the release.

----End

Performing Verification

Deploy a workload and configure the newly deployed Nginx Ingress Controller to provide network access for the workload.

Step 1 Create an Nginx workload.

1. Log in to the CCE console, click the created cluster, choose **Workloads** in the navigation pane, and click **Create Workload** in the upper right corner.
2. Enter the following information and click **OK**.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx    # If an image from an open-source image registry is used, enter the image name. If you use an image in My Images, obtain the image path from SWR.
        imagePullPolicy: Always
        name: nginx
      imagePullSecrets:
      - name: default-secret
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx
  name: nginx
spec:
```



```
  ports:
  - name: service0
    port: 80                # Port for accessing a Service.
    protocol: TCP           # Protocol used for accessing a Service. The value can be TCP or UDP.
    targetPort: 80          # Port used by the service to access the target container. In this example, the
Nginx image uses port 80 by default.
  selector:                 # Label selector. A Service selects a pod based on the label and forwards the
requests for accessing the Service to the pod.
    app: nginx
  type: ClusterIP           # Type of a Service. ClusterIP indicates that a Service is only reachable from
within the cluster.
```

Step 2 Create an ingress and use the newly deployed Nginx Ingress Controller to provide network access.

1. In the navigation pane, choose **Services & Ingresses**. Click the **Ingresses** tab and click **Create from YAML** in the upper right corner.

   NOTE

   When interconnecting with Nginx Ingress Controller that is not deployed using an add-on, you can create an ingress only through YAML.

2. Enter the following information and click **OK**.

   For clusters of v1.23 or later:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
spec:
  ingressClassName: ccedemo  # Enter the ingressClass of the newly created Nginx Ingress Controller.
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /
        pathType: ImplementationSpecific   # The matching depends on IngressClass.
        backend:
          service:
            name: nginx    # Replace it with the name of your target Service.
            port:
              number: 80   # Replace it with the port of your target Service.
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```

   For clusters earlier than v1.23:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tomcat-t1
  namespace: test
  annotations:
    kubernetes.io/ingress.class: ccedemo  # Enter the ingressClass of the newly created Nginx Ingress
Controller.
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - path: /
            pathType: ImplementationSpecific
            backend:
              serviceName: nginx  # Replace it with the name of your target Service.
              servicePort: 80     # Replace it with the port of your target Service.
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
```


Step 3 Log in to the cluster node and access the application through the Controller in the nginx-ingress add-on of the cluster and the newly deployed Nginx Ingress Controller service, respectively.

- Use the new Nginx Ingress Controller service to access the application (the Nginx page is expected to be displayed). **192.168.114.60** is the ELB address of the new Nginx Ingress Controller service.

  ```
  curl -H "Host: foo.bar.com" http://192.168.114.60
  ```

  ```
  [root@192-168-9-72 paas]# curl -H "Host: foo.bar.com" http://192.168.114.60
  <!DOCTYPE html>
  <html>
  <head>
  <title>Welcome to nginx!</title>
  <style>
      body {
          width: 35em;
          margin: 0 auto;
          font-family: Tahoma, Verdana, Arial, sans-serif;
      }
  </style>
  </head>
  <body>
  <h1>Welcome to nginx!</h1>
  <p>If you see this page, the nginx web server is successfully installed and
  working. Further configuration is required.</p>

  <p>For online documentation and support please refer to
  <a href="http://nginx.org/">nginx.org</a>.<br/>
  Commercial support is available at
  <a href="http://nginx.com/">nginx.com</a>.</p>

  <p><em>Thank you for using nginx.</em></p>
  </body>
  </html>
  ```

- Use the Controller service in the nginx-ingress add-on (404 is expected to be returned). **192.168.9.226** is the ELB address of the nginx-ingress add-on.

  ```
  curl -H "Host: foo.bar.com" http://192.168.9.226
  ```

  ```
  [root@192-168-9-72 paas]# curl -H "Host: foo.bar.com" http://192.168.9.226
  default backend - 404[root@192-168-9-72 paas]#
  ```

----End

Parameter Description

Table 10-9 nginx-ingress parameters

Parameter	Description
controller.image.repository	ingress-nginx image address. It is recommended that this parameter be set to the same as the nginx-ingress add-on image provided by CCE. You can also customize the parameter. • nginx-ingress add-on image: You can view its image path in the YAML file of the installed add-on. • Custom: The custom path must ensure that the image can be pulled.
controller.image.registry	Domain name of the image repository. This parameter must be set together with **controller.image.image**. If **controller.image.repository** has been set, you do not need to set this parameter. You are advised to leave **controller.image.registry** and **controller.image.image** empty.
controller.image.image	Image name. This parameter must be set together with **controller.image.registry**. If **controller.image.repository** has been set, you do not need to set this parameter. You are advised to leave **controller.image.registry** and **controller.image.image** empty.
controller.image.tag	ingress-nginx image version. It is recommended that this parameter be set to the same as the nginx-ingress add-on image provided by CCE. You can also customize the image. The image version of the nginx-ingress add-on can be viewed in the YAML file of the installed add-on and needs to be replaced based on the add-on version.
controller.ingressClass	Specifies the name of the IngressClass of the Ingress Controller. **NOTE** The name of each Ingress Controller in the same cluster must be unique and cannot be set to **nginx** or **cce**. **nginx** is the default listening identifier of Nginx Ingress Controller in the cluster, and **cce** is the configuration of ELB Ingress Controller. Example: **ccedemo**
controller.image.digest	You are advised to leave this parameter empty. If this parameter is specified, pulling the nginx- ingress add-on image provided by CCE may fail.

Parameter	Description
controller.ingressClassResource.name	The parameter value must be the same as that of ingressClass. Example: **ccedemo**
controller.ingressClassResource.controllerValue	The listening identifier of each Ingress Controller in the same cluster must be unique and cannot be set to **k8s.io/ingress-nginx**, which is the default listening identifier of Nginx Ingress Controller. Example: **k8s.io/ingress-nginx-demo**
controller.config	Nginx configuration parameter. For details, see **Community Documents**. Parameter settings out of the range do not take effect. You are advised to add the following configurations: "keep-alive-requests": "100"
controller.extraInitContainers	init container, which is executed before the main container is started and can be used to initialize pod parameters. For details about parameter configuration examples, see **Parameter Optimization in High-Concurrency Scenarios**.
controller.admissionWebhooks.enabled	Specifies whether to enable admissionWebhooks to verify the validity of ingress objects. This prevents ingress-controller from continuously reloading resources due to incorrect configurations, which may cause service interruption. Set this parameter to **false**, indicating that the function is disabled. To enable this function, see the example in **admissionWebhook Configuration**.
controller.admissionWebhooks.patch.enabled	Specifies whether to enable admissionWebhooks. Set this parameter to **false**.
controller.service.annotations	A key-value pair. The ELB ID needs to be added, as shown in the following: kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0 For dedicated load balancers, add **elb.class** as follows: kubernetes.io/elb.class: performance
controller.resources.requests.cpu	Specifies the quantity of CPU resources requested by the Nginx controller. This parameter can be customized.
controller.resources.requests.memory	Specifies the quantity of memory resources requested by the Nginx controller. This parameter can be customized.

Parameter	Description
defaultBackend.image.repository	default-backend image path. It is recommended that this parameter be set to the same as the nginx-ingress add-on image provided by CCE. You can also customize the parameter. • nginx-ingress add-on image: You can view its image path in the YAML file of the installed add-on. • Custom: The custom path must ensure that the image can be pulled.
defaultBackend.image.tag	default-backend image version. It is recommended that this parameter be set to the same as the nginx-ingress add-on image provided by CCE. You can also customize the image.

For details about more parameters, see **ingress-nginx**.

10.9.2 Advanced Configuration of Nginx Ingress Controller

Parameter Optimization in High-Concurrency Scenarios

In high-concurrency scenarios, you can configure parameters for optimization in either of the following ways:

1. Use ConfigMap to optimize the overall parameters of Nginx Ingress Controller.
2. Use InitContainers to optimize the kernel parameters of Nginx Ingress Controller.

The optimized **value.yaml** configuration file is as follows:

```
controller:
  image:
    repository: registry.k8s.io/ingress-nginx/controller
    registry: ""
    image: ""
    tag: "v1.5.1"  # Controller version
    digest: ""
  ingressClassResource:
    name: ccedemo        # The name of each Ingress Controller in the same cluster must be unique and cannot be nginx or cce.
    controllerValue: "k8s.io/ingress-nginx-demo"  # The listening identifier of each Ingress Controller in the same cluster must be unique and cannot be set to k8s.io/ingress-nginx.
  ingressClass: ccedemo   # The name of each Ingress Controller in the same cluster must be unique and cannot be nginx or cce.
  service:
    annotations:
      kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0     #ELB ID
    kubernetes.io/elb.class: performance  # This annotation is required only for dedicated load balancers.
# Nginx parameter optimization
  config:
    keep-alive-requests: 10000
    upstream-keepalive-connections: 200
    max-worker-connections: 65536
# Kernel parameter optimization
  extraInitContainers:
```



```
  - name: init-myservice
    image: busybox
    securityContext:
      privileged: true
    command: ['sh', '-c', 'sysctl -w net.core.somaxconn=65535;sysctl -w
net.ipv4.ip_local_port_range="1024 65535"']
  extraVolumeMounts: # Mount the /etc/localtime file on the node to synchronize the time zone.
    - name: localtime
      mountPath: /etc/localtime
      readOnly: true
  extraVolumes:
    - name: localtime
      type: Hostpath
      hostPath:
        path: /etc/localtime
  admissionWebhooks: # Disable webhook authentication.
    enabled: false
    patch:
      enabled: false
  resources: # Set the controller's resource limit, which can be customized.
    requests:
      cpu: 200m
      memory: 200Mi
defaultBackend: # Set defaultBackend.
  enabled: true
  image:
    repository: registry.k8s.io/defaultbackend-amd64
    registry: ""
    image: ""
    tag: "1.5"
    digest: ""
```

admissionWebhook Configuration

Nginx Ingress Controller supports admissionWebhook configuration. You can configure the **controller.admissionWebhook** parameter to verify the validity of ingress objects. This prevents ingress-controller from continuously reloading resources due to incorrect configuration, which may cause service interruption.

NOTE

- When the admissionWebhook feature is used, webhook-related configurations must be enabled on the API server, including MutatingAdmissionWebhook and ValidatingAdmissionWebhook.

  The feature switch is --admission-control=MutatingAdmissionWebhook,ValidatingAdmissionWebhook.

  If it is not enabled, submit a service ticket to enable it.
- After admissionWebhook is enabled, if you need to uninstall and reinstall Nginx Ingress Controller, residual secrets exist and need to be manually cleared.

The **value.yaml** configuration file for enabling admissionWebhook is as follows:

```
controller:
  image:
    repository: registry.k8s.io/ingress-nginx/controller
    registry: ""
    image: ""
    tag: "v1.5.1"  # Controller version
    digest: ""
  ingressClassResource:
    name: ccedemo         # The name of each Ingress Controller in the same cluster must be unique and
cannot be nginx or cce.
    controllerValue: "k8s.io/ingress-nginx-demo"  # The listening identifier of each Ingress Controller in the
same cluster must be unique and cannot be set to k8s.io/ingress-nginx.
  ingressClass: ccedemo   # The name of each Ingress Controller in the same cluster must be unique and
```



```
cannot be nginx or cce.
  service:
    annotations:
      kubernetes.io/elb.id: 5083f225-9bf8-48fa-9c8b-67bd9693c4c0     #ELB ID
    kubernetes.io/elb.class: performance  # This annotation is required only for dedicated load balancers.
  config:
    keep-alive-requests: 100
  extraVolumeMounts: # Mount the /etc/localtime file on the node to synchronize the time zone.
    - name: localtime
      mountPath: /etc/localtime
      readOnly: true
  extraVolumes:
    - name: localtime
      type: Hostpath
      hostPath:
        path: /etc/localtime
  admissionWebhooks:
    annotations: {}
    enabled: true
    extraEnvs: []
    failurePolicy: Fail
    port: 8443
    certificate: "/usr/local/certificates/cert"
    key: "/usr/local/certificates/key"
    namespaceSelector: {}
    objectSelector: {}
    labels: {}
    existingPsp: ""
    networkPolicyEnabled: false
    service:
      annotations: {}
      externalIPs: []
      loadBalancerSourceRanges: []
      servicePort: 443
      type: ClusterIP
    createSecretJob:
      resources:           #Annotation{}
        limits:
          cpu: 20m
          memory: 40Mi
        requests:
          cpu: 10m
          memory: 20Mi
    patchWebhookJob:
      resources: {}
    patch:
      enabled: true
      image:
        registry: registry.k8s.io      #registry.k8s.io is the image repository of the webhook official website. Replace it with the address of the repository where the image is located.
        image: ingress-nginx/kube-webhook-certgen          # webhook image
        tag: v1.1.1
        digest: ""
        pullPolicy: IfNotPresent
      priorityClassName: ""
      podAnnotations: {}
      nodeSelector:
        kubernetes.io/os: linux
      tolerations: []
      labels: {}
      securityContext:
        runAsNonRoot: true
        runAsUser: 2000
        fsGroup: 2000
  resources: # Set the controller's resource limit, which can be customized.
    requests:
      cpu: 200m
      memory: 200Mi
defaultBackend: # Set defaultBackend.
```



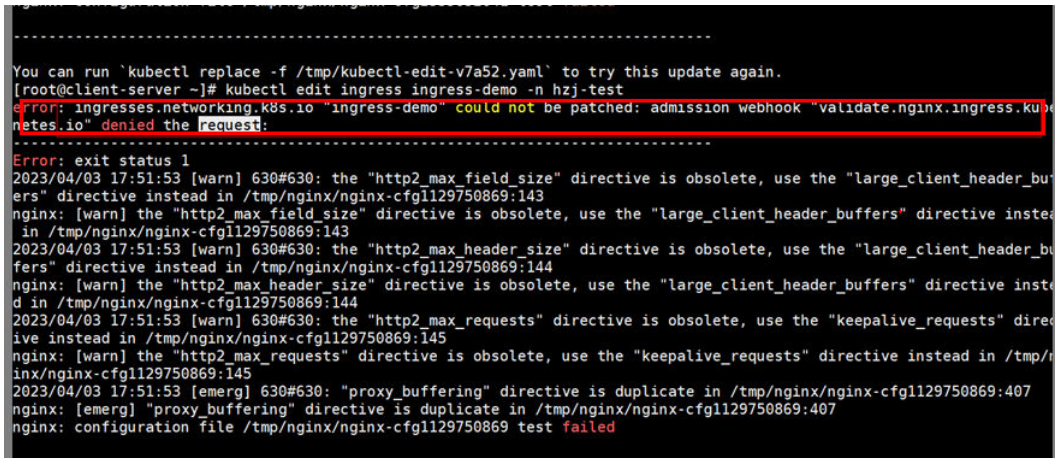
```
enabled: true
image:
  repository: registry.k8s.io/defaultbackend-amd64
  registry: ""
  image: ""
  tag: "1.5"
  digest: ""
```

Check whether admissionWebhook is verified when incorrect annotations are configured for the ingress.

For example, configure the following incorrect annotations for the ingress:

```
...
 annotations:
   nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream: "false"
   nginx.ingress.kubernetes.io/auth-tls-verify-client: optional
   nginx.ingress.kubernetes.io/auth-tls-verify-depth: "1"
...
```

When the ingress service is created, the following interception information is displayed:



10.10 CoreDNS Configuration Optimization

10.10.1 Overview

Application Scenarios

DNS is one of the important basic services in Kubernetes. When the container DNS policy is not properly configured and the cluster scale is large, DNS resolution may time out or fail. In extreme cases, a large number of services in the cluster may fail to be resolved. This section describes the best practices of CoreDNS configuration optimization in Kubernetes clusters to help you avoid such problems.

Solution

CoreDNS configuration optimization includes clients and servers.

On the client, you can optimize domain name resolution requests to reduce resolution latency, and use proper container images and NodeLocal DNSCache to reduce resolution exceptions.

- **Optimizing Domain Name Resolution Requests**
- **Selecting a Proper Image**
- **Avoiding Occasional DNS Resolution Timeout Caused by IPVS Defects**
- **Using NodeLocal DNSCache**
- **Upgrading the CoreDNS in the Cluster Timely**
- **Adjusting the DNS Configuration of the VPC and VM**

On the server, you can adjust the CoreDNS deployment status or CoreDNS configuration to improve the availability and throughput of CoreDNS in the cluster.

- **Monitoring the coredns Add-on**
- **Adjusting the coredns Deployment Status**
- **Configuring CoreDNS**

For more information about CoreDNS configurations, see **https://coredns.io/**.

CoreDNS open source community: **https://github.com/coredns/coredns**

Prerequisites

- A CCE cluster has been created. For details, see **Creating a Kubernetes Cluster**.
- You have connected to the cluster using kubectl. For details, see **Connecting to a Cluster Using kubectl**.
- The coredns add-on has been installed (the latest version is recommended). For details, see **CoreDNS**.

10.10.2 Client

10.10.2.1 Optimizing Domain Name Resolution Requests

DNS resolution is frequently used in Kubernetes clusters. Based on the characteristics of DNS resolution in Kubernetes, you can optimize domain name resolution requests in the following ways.

Using a Connection Pool

When a containerized application needs to frequently request another service, you are advised to use a connection pool. The connection pool can cache the link information of the upstream service to avoid the overhead of DNS resolution and TCP link reestablishment for each access.

Optimizing the resolve.conf File in the Container

The **ndots** and **search** parameters in the **resolve.conf** file determine the domain name resolution efficiency. For details about the two parameters, see **DNS Configuration**.

Optimizing the Domain Name Configuration

When a container needs to access a domain name, configure the domain name based on the following rules to improve the domain name resolution efficiency.

1. When a pod accesses a Service in the same namespace, use ***<service-name>***, which indicates the Service name.
2. When a pod accesses a Service across namespaces, use ***<service-name>.<namespace-name>***. ***namespace-name*** indicates the namespace where the Service is located.
3. When a pod accesses an external domain name of a cluster, use the FQDN domain name. This type of domain name is specified by adding a period (.) at the end of a common domain name to avoid multiple invalid search attempts caused by search domain combination. For example, to access **www.huaweicoud.com**, use the FQDN domain name **www.huaweicoud.com.**.

Using Local Cache

If the cluster specifications and the number of DNS resolution requests are large, you can cache the DNS resolution result on the node. You are advised to use NodeLocal DNSCache. For details, see **Using NodeLocal DNSCache to Improve DNS Performance**.

10.10.2.2 Selecting a Proper Image

The musl libc library of the Alpine container image differs from the standard glibc library in the following aspects:

- Alpine 3.3 and earlier versions do not support the **search** parameter. As a result, search domains cannot be specified for discovering Services.
- Multiple DNS servers configured in **/etc/resolve.conf** are concurrently requested. As a result, NodeLocal DNSCache cannot improve the DNS performance.
- When the same Socket is used to request A and AAAA records concurrently, the Conntrack source port conflict is triggered in the kernel of an earlier version. As a result, packet loss occurs.
- If the domain name cannot be resolved when Alpine is used as the base container image, update the base container image for testing.

For details about the functional differences from glibc, see **Functional differences from glibc**.

10.10.2.3 Avoiding Occasional DNS Resolution Timeout Caused by IPVS Defects

When the kube-proxy uses IPVS load balancing, you may encounter DNS resolution timeout occasionally during CoreDNS scale-in or restart. This problem is caused by a Linux kernel defect. For details, see **https://github.com/torvalds/linux/commit/35dfb013149f74c2be1ff9c78f14e6a3cd1539d1**.

You can use NodeLocal DNSCache to reduce the impact of IPVS defects. For details, see **Using NodeLocal DNSCache to Improve DNS Performance**.

10.10.2.4 Using NodeLocal DNSCache

NodeLocal DNSCache can improve the performance of service discovery. For details about NodeLocal DNSCache and how to deploy it in a cluster, see **Using NodeLocal DNSCache to Improve DNS Performance**.

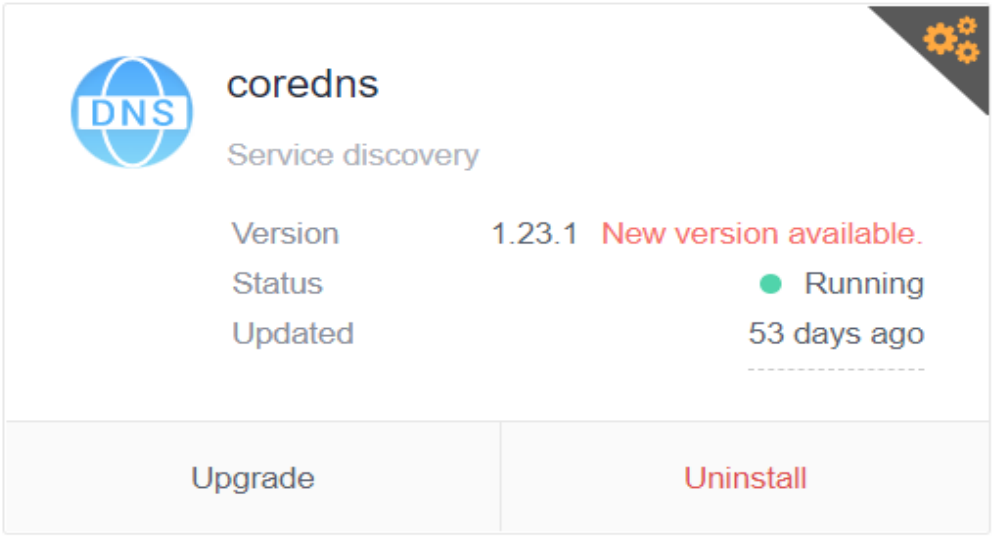
10.10.2.5 Upgrading the CoreDNS in the Cluster Timely

CoreDNS provides simple functions and is compatible with different Kubernetes versions. CCE periodically synchronizes bugs from the community and upgrades the coredns add-on. You are advised to periodically upgrade the CoreDNS. The CCE add-on management center supports the CoreDNS installation and upgrade. You can define the CoreDNS version in the cluster. If the version can be upgraded, upgrade the CoreDNS component in the cluster as soon as possible.

You can upgrade CoreDNS in a cluster by performing the following procedure:

Step 1 Log in to the CCE console, select a cluster, and click **Add-ons** in the navigation pane.

Step 2 In the **Add-on Installed**, locate the coredns add-on and click **Upgrade**.



Step 3 Set parameters as prompted. For details, see **coredns (System Resource Add-on, Mandatory)**.

----End

10.10.2.6 Adjusting the DNS Configuration of the VPC and VM

When the coredns add-on is started, it obtains the DNS configuration in the **resolve.conf** file from the deployed instance by default and uses the configuration as the upstream resolution server address. Before the coredns add-on is restarted, the **resolve.conf** configuration on the node is not reloaded. Suggestions:

- Ensure that the **resolve.conf** configuration of each node in the cluster is the same. In this way, the coredns add-on can schedule requests to any node in the cluster.

- When modifying the **resolve.conf** file, if the node has a coredns add-on, restart the coredns add-on timely to ensure status consistency.

10.10.3 Server

10.10.3.1 Monitoring the coredns Add-on

- CoreDNS exposes health metrics such as resolution results through the standard Prometheus API to detect exceptions on the CoreDNS server or even upstream DNS server.
- Port for obtaining coredns metrics. The default zone listening IP address is **{$*POD_IP*}:9153**. Retain the default value. Otherwise, Prometheus cannot collect the metrics.
- If you use on-premises Prometheus to monitor Kubernetes clusters, you can observe related metrics on Prometheus and configure alarms for the key metrics. For details, see **prometheus**.

10.10.3.2 Adjusting the coredns Deployment Status

By default, the coredns add-on is installed in the CCE cluster and the CoreDNS application runs on the same node as the service container. Pay attention to the following points when deploying the coredns add-on:

- **Adjusting the Number of CoreDNS Replicas Properly**
- **Allocating the Location of the CoreDNS Properly**
- **Manually Increasing Replicas**
- **Automatically Expanding the CoreDNS Capacity Based on the HPA**

Adjusting the Number of CoreDNS Replicas Properly

You are advised to set the number of CoreDNS replicas to at least 2 in any case and keep the number of replicas within a proper range to support the resolution of the entire cluster. The default number of instances for installing the coredns add-on in a CCE cluster is **2**.

- Modifying the number of CoreDNS replicas, CPUs, and memory size will change CoreDNS's parsing capability. Therefore, evaluate the impact before the operation.
- By default, podAntiAffinity (pod anti-affinity) is configured for the coredns add-on. If a node already has a CoreDNS pod, no new pod can be added. That is, only one CoreDNS pod can run on a node. If there are more configured CoreDNS replicas than cluster nodes, the excess pods cannot be scheduled. Therefore, keep the number of replicas less than or equal to the number of nodes.

Allocating the Location of the CoreDNS Properly

- You are advised to distribute CoreDNS replicas on nodes in different AZs and clusters to prevent single-node or single-AZ faults. By default, podAntiAffinity (pod anti-affinity) is configured for the coredns add-on. Some or all replicas may be deployed on the same node due to insufficient node resources. In this case, delete the pod and reschedule it again.

- The CPU and memory of the cluster node where the coredns add-on runs must not be used up. Otherwise, the QPS and response of domain name resolution will be affected.

Manually Increasing Replicas

You can call the API to modify the coredns add-on specifications. Modify the number of replicas as required. For details, see **Customizing CoreDNS Specifications**.

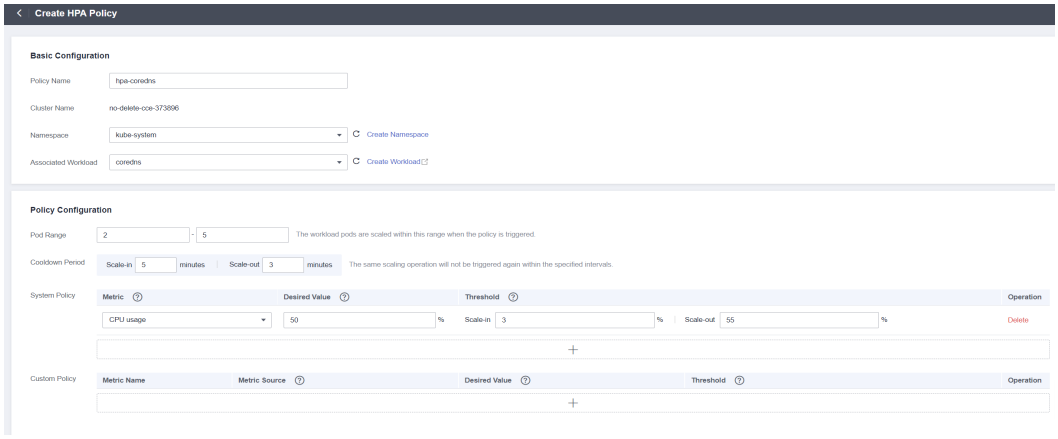
Automatically Expanding the CoreDNS Capacity Based on the HPA

HPA frequently scales down the number of the coredns add-on replicas. Therefore, you are advised not to use HPA. If HPA is required, use the CCE-developed add-on cce-hpa-controller to configure an HPA automated scale-out policy. The process is as follows:

Step 1 Log in to the CCE console and access the cluster details page. Choose **Add-ons** in the navigation pane, locate **cce-hpa-controller** on the right, and click **Install**.

Step 2 You can select **Single** or **Custom** for **Add-on Specifications**, and click **Install**. For details, see **cce-hpa-controller**.

Step 3 On the CCE console, choose **Workload** > **Scaling** in the navigation pane, select **kube-system** for the namespace, and click **Create HPA Policy**.



In the **Basic Settings** area, you can edit the policy name. Set **Namespace** to **kube-system** and associated workload to coredns.

On the **Policy Configuration** page, you can customize HPA policies based on metrics such as CPU usage and memory usage to automatically scale out the coredns add-on replicas.

Step 4 Click **Create**. If the latest status is **Started**, the policy has taken effect.

----End

10.10.3.3 Configuring CoreDNS

On the console, the CoreDNS add-on can only be configured with the preset specifications, which can satisfy most of the service requirements. In some

scenarios where there are requirements on the CoreDNS resource usage, you may need to customize the add-on specifications.

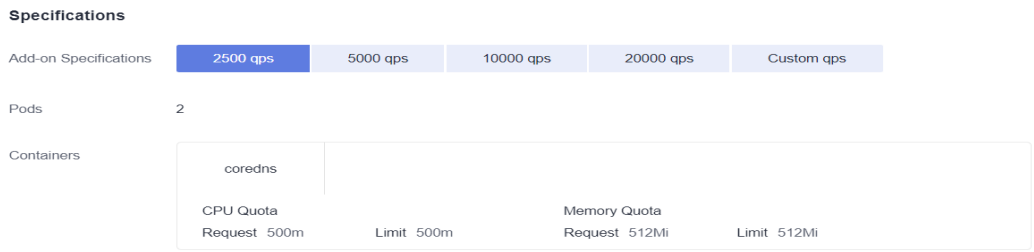
For details about how to customize CoreDNS parameters, see **Customizing CoreDNS Specifications**.

CoreDNS official document: **https://coredns.io/plugins/**

Configuring CoreDNS Specifications

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

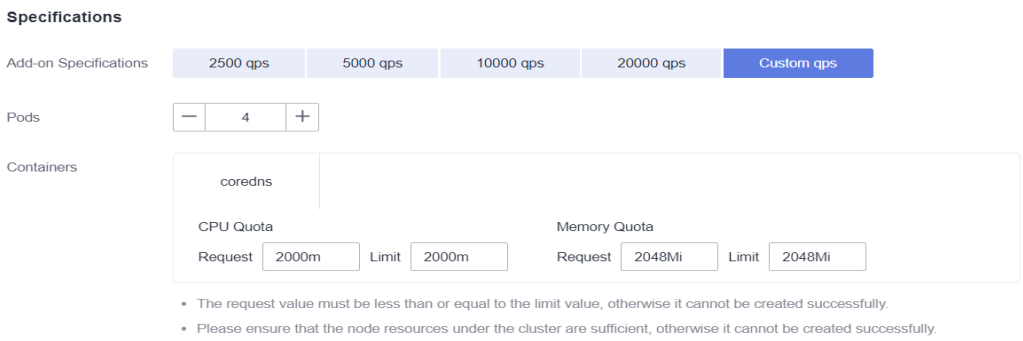
Step 2 In the navigation pane, choose **Add-ons**. On the displayed page, click **Edit** under **CoreDNS**. The add-on details page is displayed.



Step 3 In the **Specifications** area, configure coredns specifications.

Step 4 You can select the domain name resolution QPS provided by CoreDNS based on service requirements.

Step 5 You can also customize CoreDNS parameters of the cluster by specifying the number of pods, CPU quota, and memory quota.



Step 6 Click **OK**.

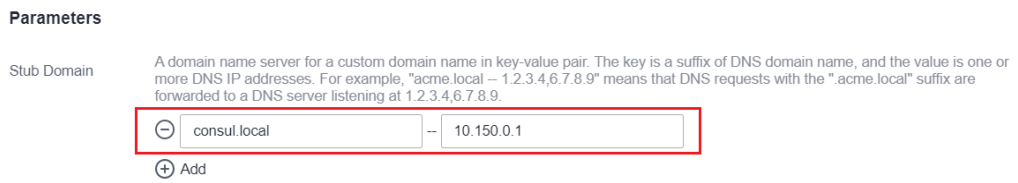
----End

Properly Configuring the Stub Domain for DNS

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 In the navigation pane, choose **Add-ons**. Then, click **Edit** under **CoreDNS**.

Step 3 Add a stub domain in the **Parameters** area. The format is a key-value pair. The key is a DNS suffix domain name, and the value is a DNS IP address or a group of DNS IP addresses, for example, **consul.local -- 10.150.0.1**.



Step 4 Click **OK**.

Step 5 Choose **ConfigMaps and Secrets** in the navigation pane, select the **kube-system** namespace, and view the ConfigMap data of **coredns** to check whether the update is successful.

The corresponding Corefile content is as follows:

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
consul.local:5353 {
    bind {$POD_IP}
    errors
    cache 30
    forward . 10.150.0.1
}
```

----End

Properly Configuring the Host

To specify hosts for a specific domain name, you can use the hosts add-on. An example is as follows:

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 In the navigation pane, choose **Add-ons**. Then, click **Edit** under **CoreDNS**.

Step 3 Edit the advanced configuration under **Parameters** and add the following content to the **plugins** field:

```
{
  "configBlock": "192.168.1.1 www.example.com\nfallthrough",
  "name": "hosts"
}
```


NOTICE

The **fallthrough** field must be configured. **fallthrough** indicates that when the domain name to be resolved cannot be found in the hosts file, the resolution task is transferred to the next CoreDNS plug-in. If **fallthrough** is not specified, the task ends and the domain name resolution stops. As a result, the domain name resolution in the cluster fails.

For details about how to configure the hosts file, visit **https://coredns.io/plugins/hosts/**.

Step 4 Click **OK**.

Step 5 Choose **ConfigMaps and Secrets** in the navigation pane, select the **kube-system** namespace, and view the ConfigMap data of **coredns** to check whether the update is successful.

The corresponding Corefile content is as follows:

```
.:5353 {
    bind {$POD_IP}
    hosts {
      192.168.1.1 www.example.com
      fallthrough
    }
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    ready {$POD_IP}:8081
}
```

----End

Configuring the Default Protocol Between the forward Plug-in and the Upstream DNS Service

Step 1 The NodeLocal DNSCache uses TCP to communicate with the CoreDNS. The CoreDNS communicates with the upstream DNS server based on the protocol used by the request source. By default, external domain name resolution requests from service containers pass through NodeLocal DNSCache and CoreDNS in sequence, and finally request the DNS server in the VPC using TCP.

Step 2 However, the cloud server does not support TCP. To use NodeLocal DNSCache, modify the CoreDNS configuration so that UDP is preferentially used to communicate with the upstream DNS server, preventing resolution exceptions. You are advised to use the following method to modify the CoreDNS configuration file:

The forward plug-in is used to set the upstream Nameservers DNS server. The following parameters are included:

prefer_udp: Even if a request is received through TCP, UDP must be used first.

If you want CoreDNS to preferentially use UDP to communicate with upstream systems, set the protocol in the forward plug-in to **prefer_udp**. For details about the forward plug-in, see **https://coredns.io/plugins/forward/**.

1. Log in to the CCE console and click the cluster name to access the cluster console.
2. In the navigation pane, choose **Add-ons**. Then, click **Edit** under **CoreDNS**.
3. Edit the advanced configuration under **Parameters** and modify the following content in the **plugins** field:

```
{
    "configBlock": "prefer_udp",
    "name": "forward",
    "parameters": ". /etc/resolv.conf"
}
```

----End

Configuring IPv6 Resolution Properly

If the IPv6 kernel module is not disabled on the Kubernetes cluster host machine, the container initiates IPv4 and IPv6 resolution at the same time by default when requesting the coredns add-on. Generally, only IPv4 addresses are used. Therefore, if you only configure **DOMAIN in IPv4 address**, the coredns add-on forwards the request to the upstream DNS server for resolution because the local configuration cannot be found. As a result, the DNS resolution request of the container slows down.

CoreDNS provides the template plug-in. After being configured, CoreDNS can immediately return an empty response to all IPv6 requests to prevent the requests from being forwarded to the upstream DNS.

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 In the navigation pane, choose **Add-ons**. On the displayed page, click **Edit** under **CoreDNS**.

Step 3 Edit the advanced configuration under **Parameters** and add the following content to the **plugins** field:

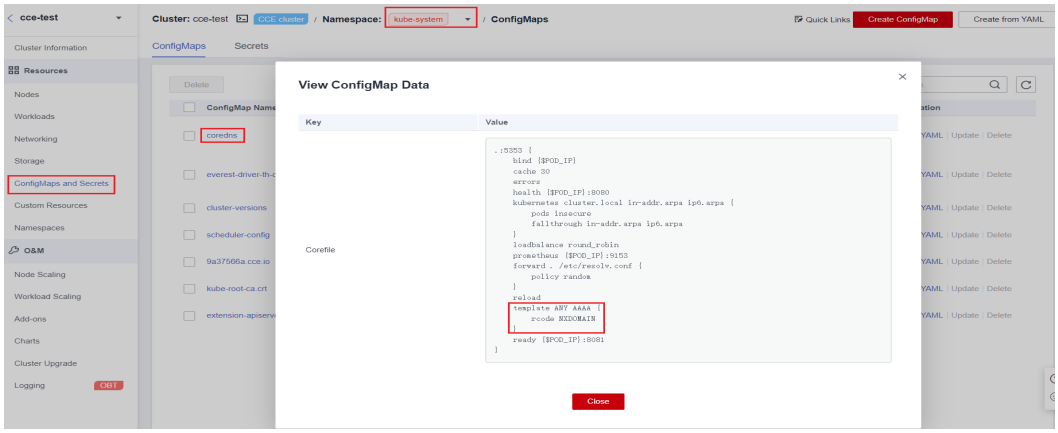
- AAAA indicates an IPv6 resolution request. If **NXDOMAIN** is returned in the **rcode** control response, meaning that no resolution result is returned.

For details about the template plug-in, visit **https://github.com/coredns/coredns/tree/master/plugin/template**.

```
{
  "configBlock": "rcode NXDOMAIN",
  "name": "template",
  "parameters": "ANY AAAA"
}
```

Step 4 Click **OK**.

Step 5 In the navigation pane, choose **ConfigMaps and Secrets**. In the **kube-system** namespace, view the coredns configuration data to check whether the update is successful.



Corresponding Corefile content:

```
.:5353 {
    bind {$POD_IP}
    cache 30
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
    template ANY AAAA {
        rcode NXDOMAIN
    }
    ready {$POD_IP}:8081
}
```

----End

Properly Configuring Cache Policies

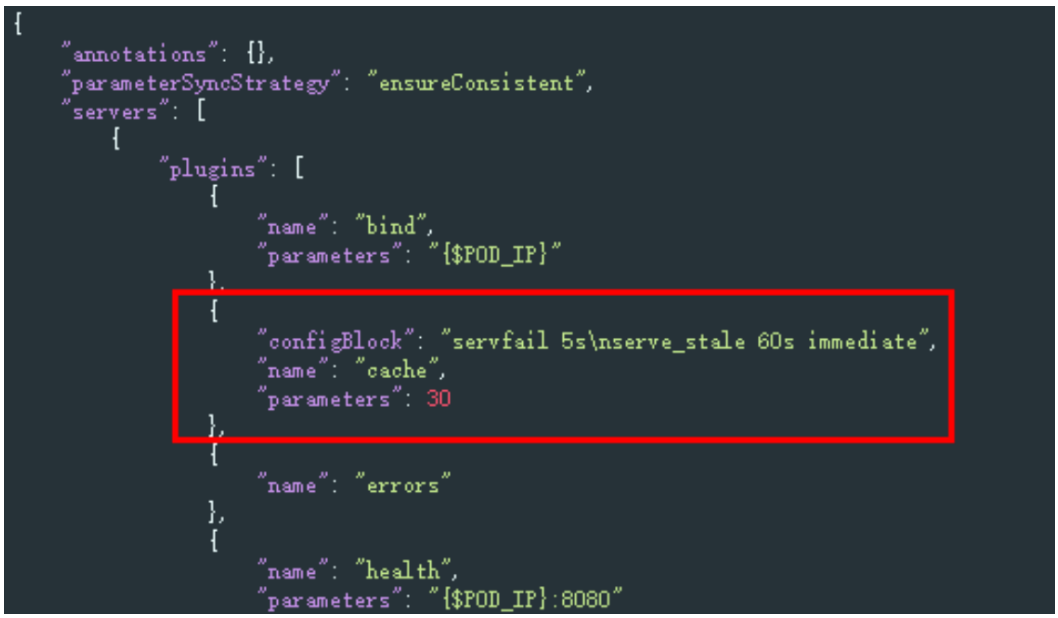
If you configure CoreDNS with an upstream DNS server, you can implement a cache policy that enables CoreDNS to use the expired local cache when it is unable to access the upstream DNS server.

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 In the navigation pane, choose **Add-ons**. On the displayed page, click **Edit** under **CoreDNS**.

Step 3 In the window that slides out from the right, in the **Parameters** area, modify the cache content in the **plugins** field for **Advance Config**. For details about how to configure the cache, see **https://coredns.io/plugins/cache/**.

```
{
    "configBlock": "servfail 5s\nserve_stale 60s immediate",
    "name": "cache",
    "parameters": 30
}
```



Step 4 Click **OK**.

Step 5 In the navigation pane, choose **ConfigMaps and Secrets**. Select the **kube-system** namespace, view the data of the ConfigMap named **coredns** to check whether the update is successful.

Corresponding Corefile content:

```
.:5353 {
    bind {$POD_IP}
    cache 30 {
        servfail 5s
        serve_stale 60s immediate
    }
    errors
    health {$POD_IP}:8080
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    loadbalance round_robin
    prometheus {$POD_IP}:9153
    forward . /etc/resolv.conf {
        policy random
    }
    reload
```



```
    ready {$POD_IP}:8081
}
```

----End

10.11 Pre-Binding Container ENI for CCE Turbo Clusters

In the Cloud Native 2.0 network model, each pod is allocated an ENI or a sub-ENI (called container ENI). The speed of ENI creation and binding is slower than that of pod scaling, severely affecting the container startup speed in large-scale batch creation. Therefore, the Cloud Native Network 2.0 model provides the dynamic pre-binding of container ENIs to accelerate pod startup while improving IP resource utilization.

Constraints

- CCE Turbo clusters of 1.19.16-r4, 1.21.7-r0, 1.23.5-r0, 1.25.1-r0, or later support ENI pre-binding, global configuration at the cluster level, and custom settings at the node pool level. Custom settings of nodes out of a node pool is not supported.
- CCE Turbo clusters of 1.19.16-r2, 1.21.5-r0, 1.23.3-r0 to 1.19.16-r4, 1.21.7-r0, 1.23.5-r0 only support two parameters, **nic-minimum-target** and **nic-warm-target**, and do not support custom settings at the node pool level.
- Modify the dynamic pre-binding parameters using the console or API instead of the node annotations in the background. Otherwise, the modified annotations will be overwritten by the original values after the cluster is upgraded.
- CCE Turbo clusters of 1.19.16-r4, 1.21.7-r0, 1.23.5-r0, 1.25.1-r0, or earlier support high and low threshold for ENI buffers. If users have enabled this feature, the original high and low threshold for ENI pre-binding parameters is automatically converted to the dynamic pre-binding parameters of the container ENIs. If you want to modify the dynamic pre-binding parameters of the container ENIs on the console, change the original high and low threshold to **0:0** on the cluster configuration management console.
- For BMS node pools in CCE Turbo clusters earlier than 1.19.16-r4, 1.21.7-r0, 1.23.5-r0, and 1.25.1-r0, the high and low thresholds for ENI pre-binding (0.3:0.6 by default) are used by default. After the cluster is upgraded, the original high and low threshold still takes effect. You are advised to convert the high and low threshold parameters to the dynamic pre-binding parameters of container ENIs on the configuration management console of the node pool and delete the high and low threshold configuration to enable the latest dynamic pre-binding parameters.
- In the non-node pool BMS scenario of CCE Turbo, clusters of version earlier than 1.19.16-r4, 1.21.7-r0, 1.23.5-r0 and 1.25.1-r0 use the high and low threshold for ENI buffers (0.3:0.6 by default) by default. After the cluster is upgraded, the original high and low threshold still takes effect. If you want to enable the cluster-level global configuration, delete the annotation (**node.yangtse.io/eni-warm-policy**) of the node in the background.

How It Works

CCE Turbo provides four dynamic pre-binding parameters for container ENIs. You can properly configure the parameters based on your service requirements. (The

node pool-level dynamic ENI pre-binding parameters take priority over the cluster-level dynamic ENI pre-binding parameters.)

Table 10-10 Parameters of the dynamic ENI pre-binding policy

Parameter	Default Value	Description	Suggestion
nic-minimum-target	10	Minimum number of container ENIs bound to a node. The parameter value must be a positive integer. The value **10** indicates that there are at least 10 container ENIs bound to a node. If the number you entered exceeds the container ENI quota of the node, the ENI quota will be used.	Configure these parameters based on the number of pods.
nic-maximum-target	0	If the number of ENIs bound to a node exceeds the value of **nic-maximum-target**, the system does not proactively pre-bind ENIs. If the value of this parameter is greater than or equal to the value of **nic-minimum-target**, the check on the maximum number of the pre-bound ENIs is enabled. Otherwise, the check is disabled. The parameter value must be a positive integer. The value **0** indicates that the check on the upper limit of pre-bound container ENIs is disabled. If the number you entered exceeds the container ENI quota of the node, the ENI quota will be used.	Configure these parameters based on the number of pods.
nic-warm-target	2	Minimum number of pre-bound ENIs on a node. The value must be a number. When the value of **nic-warm-target** + the number of bound ENIs is greater than the value of **nic-maximum-target**, the system will pre-bind ENIs based on the difference between the value of **nic-maximum-target** and the number of bound ENIs.	Set this parameter to the number of pods that can be scaled out instantaneously within 10 seconds.

Parameter	Default Value	Description	Suggestion
nic-max-above-warm-target	2	Only when the number of idle ENIs on a node minus the value of **nic-warm-target** is greater than the threshold, the pre-bound ENIs will be unbound and reclaimed. The value can only be a number. • Setting a larger value of this parameter slows down the recycling of idle ENIs and accelerates pod startup. However, the IP address usage decreases, especially when IP addresses are insufficient. Therefore, **exercise caution when increasing the value of this parameter**. • Setting a smaller value of this parameter accelerates the recycling of idle ENIs and improves the IP address usage. However, when a large number of pods increase instantaneously, the startup of some pods slows down.	Set this parameter based on the difference between the number of pods that are frequently scaled on most nodes within minutes and the number of pods that are instantly scaled out on most nodes within 10 seconds.

Configuration Example

Level	Service Scenario	Configuration Example
Cluster	All nodes use the c7.4xlarge.2 model (sub-ENI quota: 128). Most nodes run about 20 pods. Most nodes can run a maximum of 60 pods. Most nodes can scale out 10 pods within 10 seconds. Most nodes frequently scale in or out 15 pods within minutes.	Cluster-level global configuration: • nic-minimum-target: 20 • nic-maximum-target: 60 • nic-warm-target: 10 • nic-max-above-warm-target: 5

Level	Service Scenario	Configuration Example
Node pool	A node pool that uses the c7.8xlarge.2 high-specification model is created in the cluster. (sub-ENI quota: 256) Most nodes run about 100 pods. Most nodes can run a maximum of 128 pods. Most nodes can scale out 10 pods within 10 seconds. Most nodes frequently scale in or out 12 pods within minutes.	Custom settings at the node pool level: • nic-minimum-target: 100 • nic-maximum-target: 120 • nic-warm-target: 10 • nic-max-above-warm-target: 2

NOTE

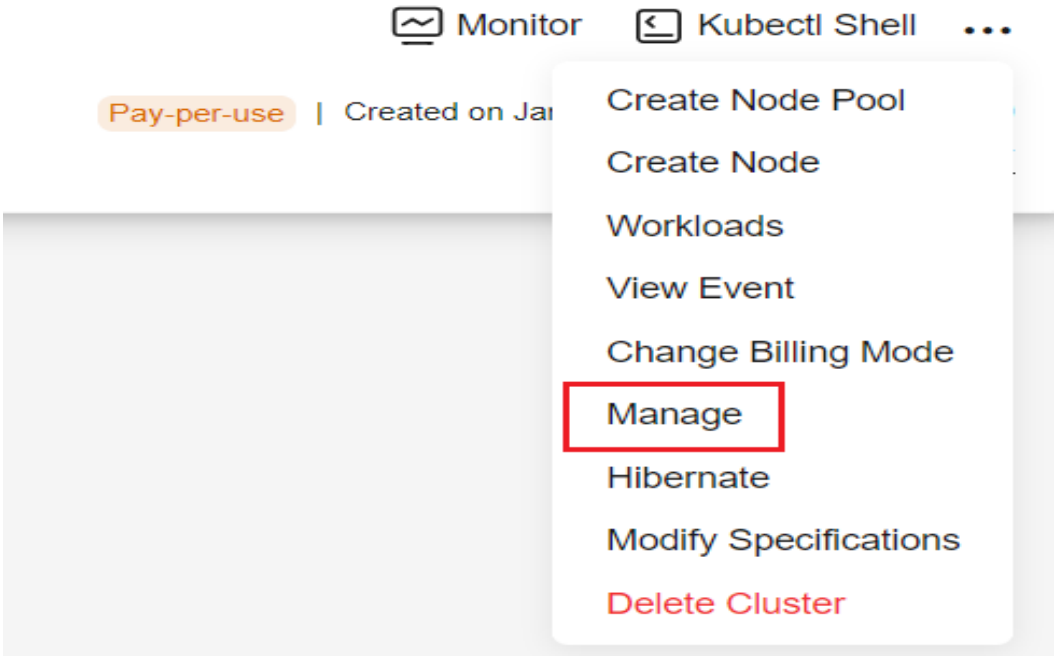
Pods using HostNetwork are excluded.

Cluster-level Global Configuration

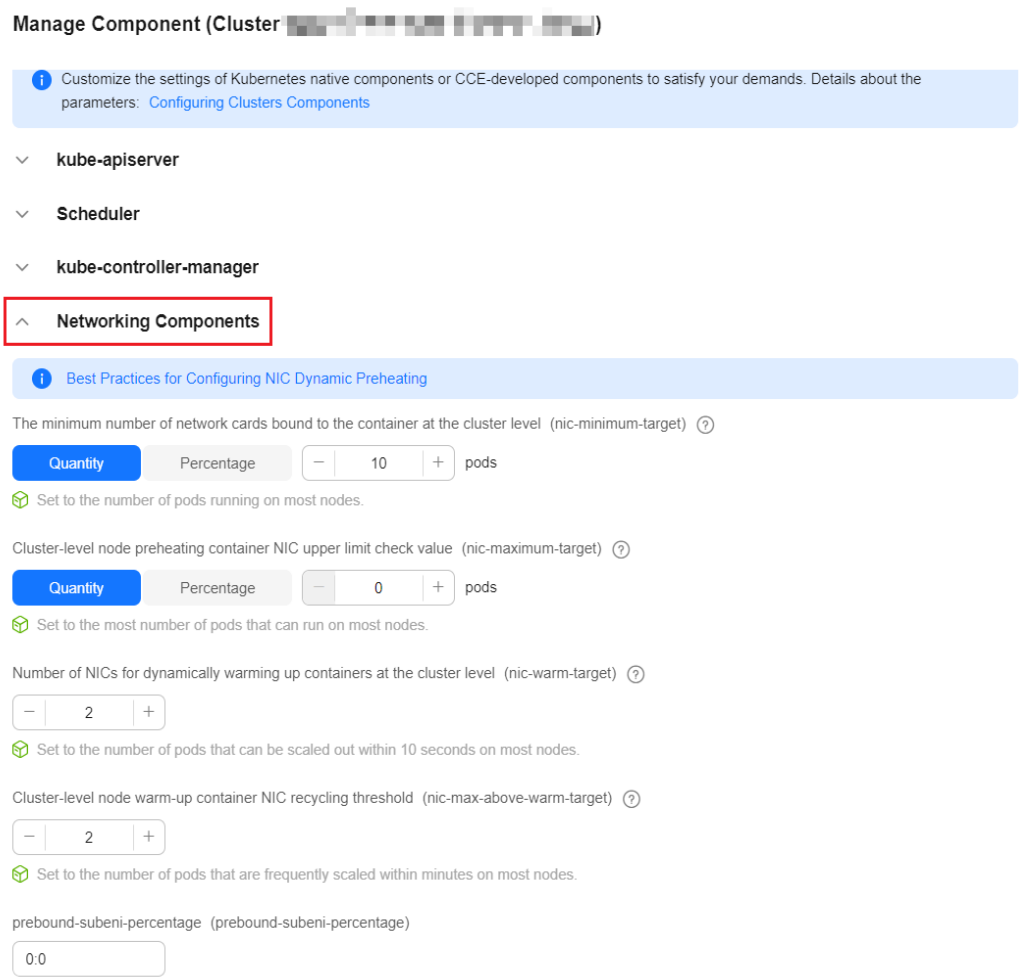
Step 1 Log in to the CCE console. In the navigation pane, choose **Clusters**.

Step 2 Click ··· next to the target cluster and choose **Manage**.

Figure 10-37 Managing a cluster



Step 3 In the window that slides out from the right, click **Networking Components**. For details about the parameter configurations, see **Configuration Example**.



Step 4 After the configuration is complete, click **OK**. Wait for about 10 seconds for the configuration to take effect.

----End

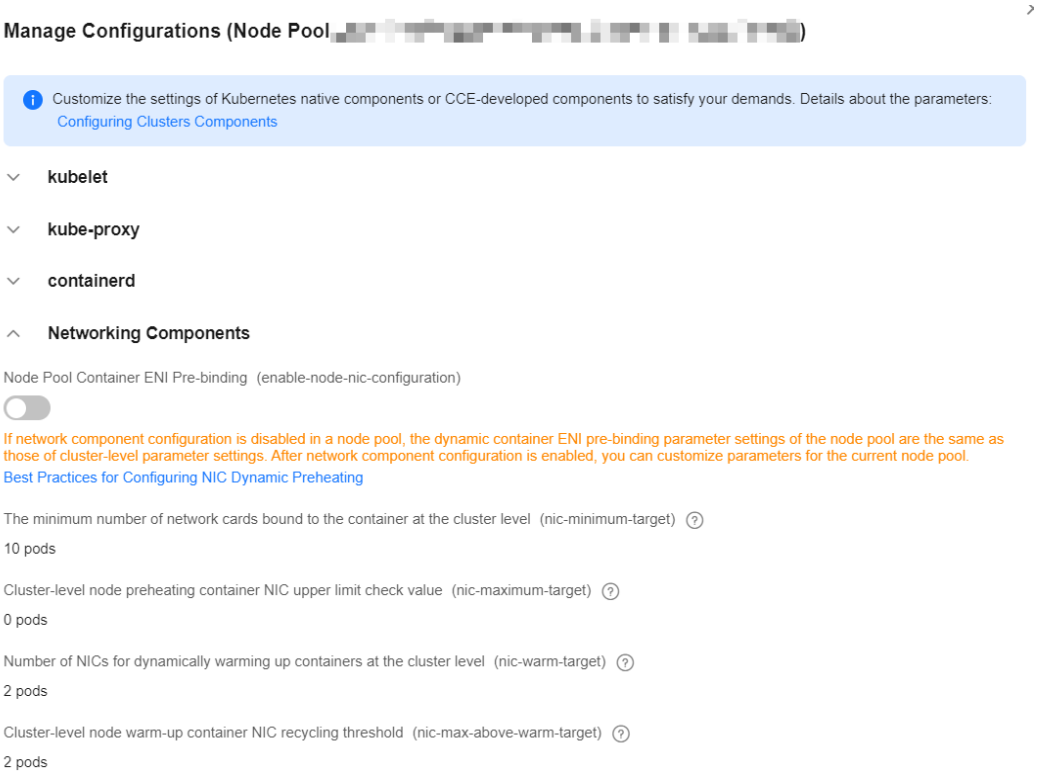
Custom Settings at the Node Pool Level

Step 1 Log in to the CCE console.

Step 2 Click the cluster name to access the cluster console, choose **Nodes** in the navigation pane, and click the **Node Pools** tab.

Step 3 Locate the row containing the target node pool and click **Manage**.

Step 4 In the window that slides out from the right, click **Networking Components** and enable node pool container ENI pre-binding. For details about the parameter configurations, see **Configuration Example**.



Step 5 After the configuration is complete, click **OK**. Wait for about 10 seconds for the configuration to take effect.

----End

10.12 Connecting a Cluster to the Peer VPC Through an Enterprise Router

Application Scenarios

An enterprise router connects virtual private clouds (VPCs) and on-premises networks to build a central hub network and implement communication between VPCs in the same region. It has high specifications, provides high bandwidth, and delivers high performance. With the enterprise routers, CCE clusters in different VPCs can access each other.

Clusters in different VPCs cannot communicate with VMs in the peer VPCs within a short period of time after containers are created in the clusters. To solve this problem, attach the peer VPCs to the enterprise routers. In a CCE Turbo cluster, configure parameters to delay the pod startup to solve this problem. For details, see **Configuring Parameters to Delay the Pod Startup in a CCE Turbo Cluster**.

Network Planning

Before attaching VPCs to the enterprise routers, determine the VPC CIDR blocks and the enterprise routers' route table. Ensure that the following requirements are met.

Resource	Description
VPC	• The VPC CIDR blocks cannot overlap. • The CIDR blocks of the VPCs are propagated to the enterprise routers' route tables. These CIDR blocks cannot be modified. Overlapping CIDR blocks may cause route conflicts. Additionally, the container CIDR blocks cannot conflict with the node CIDR blocks in the peer VPCs. Otherwise, the network is unavailable. • If your existing VPCs have overlapping CIDR blocks, manually add static routes to the route tables of the enterprise routers. The destination can be the VPC subnet CIDR blocks or smaller ones.
Enterprise router	After **Default Route Table Association** and **Default Route Table Propagation** are enabled and a VPC attachment is created, the system automatically: • Associates the VPC attachment with the default route table of the enterprise router. • Propagates the VPC attachment to the default route table of the enterprise router. The route table automatically learns the VPC CIDR blocks.

For details, see **Step 1: Plan Networks and Resources**.

Creating an Enterprise Router

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 Choose **Service List** > **Networking** > **Enterprise Router**.

Step 4 Enter the **Enterprise Router** page.

Step 5 Click **Create Enterprise Router** in the upper right corner.

Step 6 Enter the **Create Enterprise Router** page.

Step 7 Configure basic information following instructions. For details about the parameter settings, see **Table 10-11**.

Figure 10-38 Creating an enterprise router

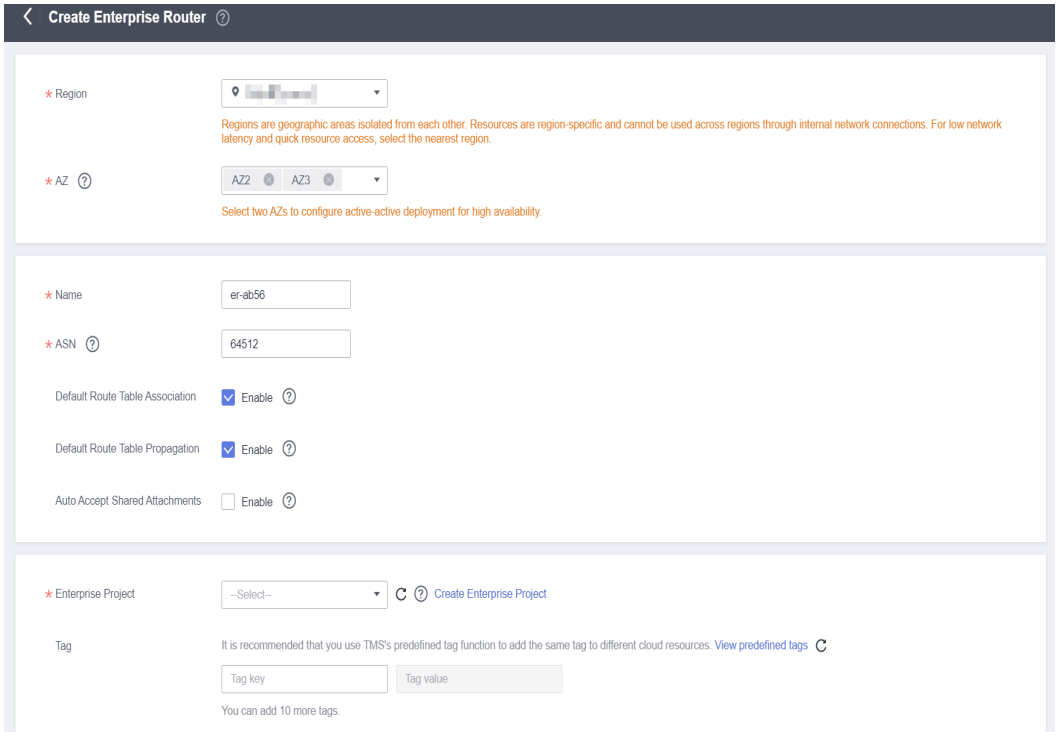


Table 10-11 Parameters for creating an enterprise router

Parameter	Setting	Example Value
Region	Select the region nearest to your target users. Once the enterprise router is created, the region cannot be changed.	CN-Hong Kong
AZ	Select two AZs to deploy your enterprise router. You can change them after the enterprise router is created.	AZ1 AZ2
Name	Specify the enterprise router name. You can change it after the enterprise router is created.	er-test-01
ASN	Enter an ASN based on your network plan. It cannot be changed after the enterprise router is created.	64512
Default Route Table Association	If you select this option, you do not need to create route tables or associations. You can change your option after the enterprise router is created.	Enable
Default Route Table Propagation	If you select this option, you do not need to create route tables, propagations, or routes. You can change your option after the enterprise router is created.	Enable

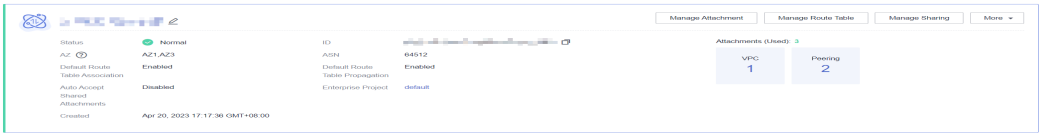
Parameter	Setting	Example Value
Auto Accept Shared Attachments	If you do not select this option, you must accept the requests for creating attachments to this enterprise router from other users with whom this enterprise router is shared.	Disable
Enterprise Project	Select an enterprise project for the enterprise router. You can change it after the enterprise router is created.	default
Tag	Add tags to help you identify your enterprise router. You can change them after the enterprise router is created.	**Tag key**: test **Tag value**: 01
Description	Provide supplementary information about the enterprise router. You can change it after the enterprise router is created.	-

Step 8 Click **Create Now**.

Step 9 Check the information on the page displayed and click **Submit**. Back to the enterprise router list.

Step 10 View the enterprise router's status. If its status changes from **Creating** to **Normal**, the enterprise router is created.

Figure 10-39 Created an enterprise router



----End

Creating a VPC Attachment to an Enterprise Router

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 Choose **Service List** > **Networking** > **Enterprise Router**. Enter the **Enterprise Router** page.

Step 4 Search for the target enterprise router by name.

Step 5 Perform either of the following operations to go to the **Attachments** tab:

- In the upper right corner of the page, click **Manage Attachment**.
- Click the name of the enterprise router and click the **Attachments** tab. For details, see **Adding VPC Attachments to an Enterprise Router**.

Step 6 Click **Create Attachment**. The **Create Attachment** page is displayed.

Step 7 Configure the parameters as prompted. For details, see **Table 10-12**.

Figure 10-40 Creating an attachment

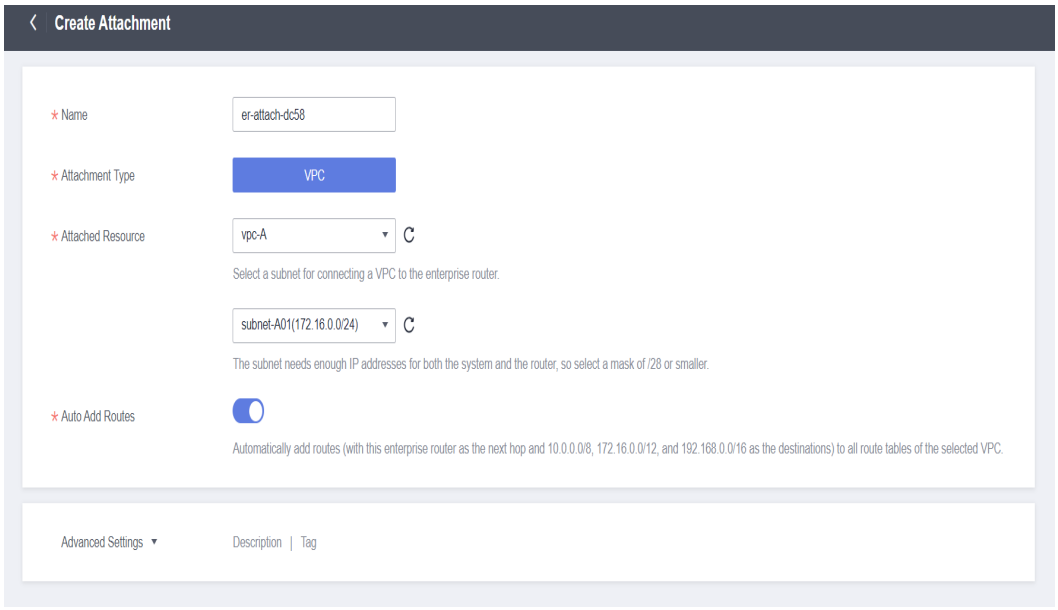


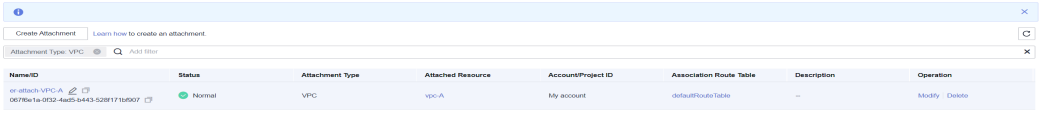
Table 10-12 Parameter description

Parameter	Setting	Example Value
Name	Specify the name of the VPC attachment. You can change it after the attachment is created.	er-attach-01
Attachment Type	Select **VPC**. The type cannot be changed after the attachment is created.	VPC
Attached Resource	1. Select the VPC to be attached to the enterprise router from the drop-down list. The VPC cannot be changed after the attachment is created. 2. Select the subnet to be attached to the enterprise router from the drop-down list. The subnet cannot be changed after the attachment is created.	• VPC: vpc-demo-01 • Subnet: subnet-demo-01

Parameter	Setting	Example Value
Auto Add Routes	• If you enable **Auto Add Routes** when creating a VPC attachment, you do not need to manually add static routes to the VPC route table. Instead, the system automatically adds routes (with this enterprise router as the next hop and 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16 as the destinations) to all route tables of the VPC. • If an existing route in the VPC route tables has a destination to 10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16, the routes will fail to be added. In this case, do not to enable **Auto Add Routes**. After the attachment is created, manually add routes. • Do not set the destination of a route (with an enterprise router as the next hop) to 0.0.0.0/0 in the VPC route table. If an ECS in the VPC has an EIP bound, the VPC route table will have a policy-based route with 0.0.0.0/0 as the destination, which has a higher priority than the route with the enterprise router as the next hop. In this case, traffic is forwarded to the EIP and cannot reach the enterprise router.	Enable
Description	Provide supplementary description about the attachment. You can change it after the attachment is created.	-
Tag	Add tags to help you identify your attachment. You can change them after the attachment is created.	**Tag key**: test **Tag value**: 01

Step 8 Click **Create Now**. If the status changes from **Creating** to **Normal**, the attachment is created. Repeat steps **Step 6** to **Step 7** to attach other VPCs.

Figure 10-41 Attaching a VPC to the enterprise router



----End

Verifying the Network

Step 1 Log in to the CCE console and search for the CCE cluster that has been attached to the VPC.

Step 2 Click the name of the target cluster. In the navigation pane, choose **Nodes** to view the IP address of the node.

Figure 10-42 Viewing the IP address of the node on the CCE console

Step 3 Log in to the node. For details, see **Logging In to a Node**. In this example, use VNC provided on the management console to log in to the ECS.

Step 4 Run the following command on the ECS console:

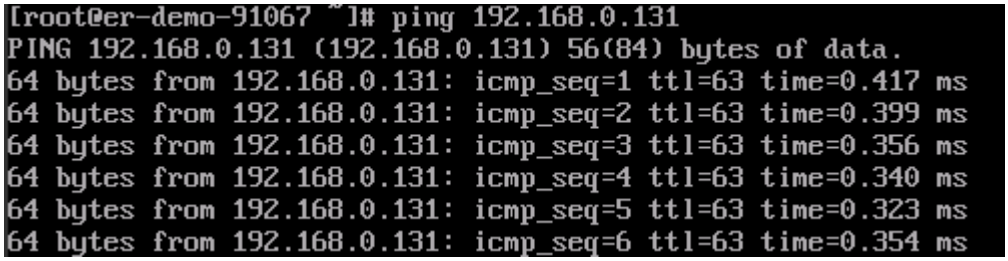
```
ping {ECS address}
```

Take the cluster in **vpc-ER-demo2** as an example. Log in to the **er-demo2-04260** node and access the **er-demo1-61379** node in the cluster in **vpc-ER-demo1**. The IP address of the node is **192.168.0.131**.

```
ping 192.168.0.131
```

If the following information is displayed, the network is accessible.

Figure 10-43 Viewing the command output

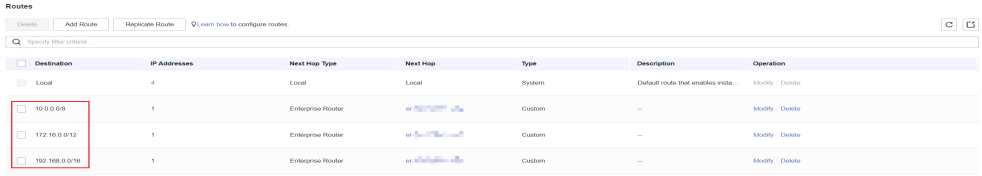


Step 5 Repeat the preceding steps to verify the communication between nodes.

If a node using the peer VPC cannot be pinged, check:

1. Whether the security group rules of the node allow the ICMP protocol.
2. Whether a CIDR block conflict occurs in the VPC route table. Note that the container CIDR blocks cannot conflict with the default CIDR blocks of the enterprise routers. For details, see **Network Planning**.

Figure 10-44 Viewing the VPC route table



----End

11 Storage

11.1 Expanding the Storage Space

The storage classes that can be expanded for CCE nodes are as follows:

Table 11-1 Capacity expansion methods

Type	Name	Purpose	Capacity Expansion Method
Node disk	System disk	A disk attached to a node for installing the operating system	**Expanding System Disk Capacity**
	Data disk	A disk that must be attached to a node for the container engine and kubelet	• **Expanding the Capacity of a Data Disk Used by Container Engines** • **Expanding the Capacity of a Data Disk Used by kubelet** • **Expanding the Capacity of a Data Disk Shared Between the Container Engine and kubelet**
Container storage	Pod container space	The base size of a container, which is, the upper limit of the disk space occupied by each pod (including the storage space occupied by container images)	**Expanding the Capacity of a Data Disk Used by Pod (basesize)**
	PVC	Storage resources mounted to the containers	**Expanding a PVC**

Expanding System Disk Capacity

EulerOS 2.9 is used as the sample OS. There is only one partition (**/dev/vda1**) with a capacity of 50 GiB in the system disk **/dev/vda**, and then 50 GiB is added to the system disk. In this example, the additional 50 GiB is allocated to the existing **/dev/vda1** partition.

Step 1 Expand the capacity of the system disk on the EVS console.

Step 2 Log in to the node and run the **growpart** command to check whether growpart has been installed.

If the tool operation guide is displayed, the growpart has been installed. Otherwise, run the following command to install growpart:

```
yum install cloud-utils-growpart
```

Step 3 Run the following command to view the total capacity of the system disk **/dev/vda**:

```
fdisk -l
```

If the following information is displayed, the total capacity of **/dev/vda** is 100 GiB.

```
[root@test-48162 ~]# fdisk -l
Disk /dev/vda: 100 GiB, 107374182400 bytes, 209715200 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x78d88f0b

Device     Boot Start       End   Sectors Size Id Type
/dev/vda1  *     2048 104857566 104855519  50G 83 Linux

Disk /dev/vdb: 100 GiB, 107374182400 bytes, 209715200 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk /dev/mapper/vgpaas-dockersys: 90 GiB, 96632569856 bytes, 188735488 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes

Disk /dev/mapper/vgpaas-kubernetes: 10 GiB, 10733223936 bytes, 20963328 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

Step 4 Run the following command to check the capacity of the system disk partition **/dev/vda1**:

```
df -TH
```

Information similar to the following is displayed:

```
[root@test-48162 ~]# df -TH
Filesystem                Type      Size  Used Avail Use% Mounted on
devtmpfs                  devtmpfs  1.8G     0  1.8G   0% /dev
tmpfs                     tmpfs     1.8G     0  1.8G   0% /dev/shm
tmpfs                     tmpfs     1.8G   13M  1.8G   1% /run
tmpfs                     tmpfs     1.8G     0  1.8G   0% /sys/fs/cgroup
/dev/vda1                 ext4       53G  3.3G   47G   7% /
tmpfs                     tmpfs     1.8G   75M  1.8G   5% /tmp
/dev/mapper/vgpaas-dockersys  ext4       95G  1.3G   89G   2% /var/lib/docker
/dev/mapper/vgpaas-kubernetes ext4       11G   39M   10G   1% /mnt/paas/kubernetes/kubelet
...
```


Step 5 Run the following command to extend the partition using growpart:

```
growpart System disk Partition number
```

The partition number is **1** because there is only one **/dev/vda1** partition in the system disk, as shown in the following command:

```
growpart /dev/vda 1
```

Information similar to the following is displayed:

```
CHANGED: partition=1 start=2048 old: size=104855519 end=104857567 new: size=209713119
end=209715167
```

Step 6 Run the following command to extend the file system:

```
resize2fs Disk partition
```

An example command is as follows:

```
resize2fs /dev/vda1
```

Information similar to the following is displayed:

```
resize2fs 1.45.6 (20-Mar-2020)
Filesystem at /dev/vda1 is mounted on /; on-line resizing required
old_desc_blocks = 7, new_desc_blocks = 13
The filesystem on /dev/vda1 is now 26214139 (4k) blocks long.
```

Step 7 Run the following command to view the new capacity of the **/dev/vda1** partition:

```
df -TH
```

Information similar to the following is displayed:

```
[root@test-48162 ~]# df -TH
Filesystem                 Type      Size  Used Avail Use% Mounted on
devtmpfs                   devtmpfs  1.8G     0  1.8G   0% /dev
tmpfs                      tmpfs     1.8G     0  1.8G   0% /dev/shm
tmpfs                      tmpfs     1.8G   13M  1.8G   1% /run
tmpfs                      tmpfs     1.8G     0  1.8G   0% /sys/fs/cgroup
/dev/vda1                  ext4      106G  3.3G   98G   4% /
tmpfs                      tmpfs     1.8G   75M  1.8G   5% /tmp
/dev/mapper/vgpaas-dockersys  ext4       95G  1.3G   89G   2% /var/lib/docker
/dev/mapper/vgpaas-kubernetes ext4       11G   39M   10G   1% /mnt/paas/kubernetes/kubelet
...
```

Step 8 Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More** > **Sync Server Data** in the row containing the target node.

----End

Expanding the Capacity of a Data Disk Used by Container Engines

CCE divides the data disk space for two parts by default. One part is used to store the Docker/containerd working directories, container images, and image metadata. The other is reserved for kubelet and emptyDir volumes. The available container engine space affects image pulls and container startup and running. This section uses Docker as an example to describe how to expand the container engine capacity.

Step 1 Expand the capacity of the data disk on the EVS console.

Step 2 Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More** > **Sync Server Data** in the row containing the target node.

Step 3 Log in to the target node.

Step 4 Run the **lsblk** command to check the block device information of the node.

A data disk is divided depending on the container storage **Rootfs**:

- Overlayfs: No independent thin pool is allocated. Image data is stored in the **dockersys** disk.

```
# lsblk
NAME                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                 8:0    0   50G  0 disk
└─vda1              8:1    0   50G  0 part /
vdb                 8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker              # Space used by the container engine
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # Space used by Kubernetes
```

  Run the following commands on the node to add the new disk capacity to the **dockersys** disk:

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

- Devicemapper: A thin pool is allocated to store image data.

```
# lsblk
NAME                          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                           8:0    0   50G  0 disk
└─vda1                        8:1    0   50G  0 part /
vdb                           8:16   0  200G  0 disk
├─vgpaas-dockersys            253:0    0   18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta       253:1    0    3G  0 lvm
│ └─vgpaas-thinpool           253:3    0   67G  0 lvm                # Space used by thinpool
│   ...
├─vgpaas-thinpool_tdata        253:2    0   67G  0 lvm
│ └─vgpaas-thinpool            253:3    0   67G  0 lvm
│   ...
└─vgpaas-kubernetes            253:4    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

  - Run the following commands on the node to add the new disk capacity to the **thinpool** disk:

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/thinpool
```

  - Run the following commands on the node to add the new disk capacity to the **dockersys** disk:

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----End

Expanding the Capacity of a Data Disk Used by kubelet

CCE divides the data disk space for container engines and pods. The container engine space stores the Docker/containerd working directories, container images, and image metadata. The other is reserved for kubelet and emptyDir volumes. To expand the kubelet space, perform the following steps:

Step 1 Expand the capacity of the data disk on the EVS console.

Step 2 Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More** > **Sync Server Data** in the row containing the target node.

Step 3 Log in to the target node.

Step 4 Run **lsblk** to view the block device information of the node.

```
# lsblk
NAME                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                 8:0    0   50G  0 disk
└─vda1              8:1    0   50G  0 part /
vdb                 8:16   0  200G  0 disk
├─vgpaas-dockersys  253:0    0   90G  0 lvm  /var/lib/docker              # Space used by the container engine
└─vgpaas-kubernetes 253:1    0   10G  0 lvm  /mnt/paas/kubernetes/kubelet  # Space used by Kubernetes
```

Step 5 Run the following commands on the node to add the new disk capacity to the Kubernetes disk:

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/kubernetes
resize2fs /dev/vgpaas/kubernetes
```

----End

Expanding the Capacity of a Data Disk Shared Between the Container Engine and kubelet

In clusters of v1.21.10-r0, v1.23.8-r0, v1.25.3-r0, and later, data disks attached to a CCE cluster can be shared disks, which is, container engines (Docker or containerd) and kubelet share the same data disk. To expand the capacity, perform the following steps:

Step 1 Expand the capacity of the data disk on the EVS console.

Step 2 Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More** > **Sync Server Data** in the row containing the target node.

Step 3 Log in to the target node.

Step 4 Run **lsblk** to view the block device information of the node.

```
# lsblk
NAME          MAJ:MIN RM SIZE RO TYPE MOUNTPOINTS
vda           253:0    0  50G  0 disk
└─vda1        253:1    0  50G  0 part /
vdb           253:16   0  60G  0 disk
└─vgpaas-share 252:0    0  60G  0 lvm  /mnt/paas # Space used by the container engine and the kubelet components
```

Step 5 Run the following commands on the node to add the new disk capacity to the shared disk:

```
pvresize /dev/vdb
lvextend -l+100%FREE -n vgpaas/share
resize2fs /dev/vgpaas/share
```

----End

Expanding the Capacity of a Data Disk Used by Pod (basesize)

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 Choose **Nodes** from the navigation pane.

Step 3 Click the Nodes tab, locate the row containing the target node, and choose **More** > **Reset Node** in the **Operation** column.

NOTICE

Resetting a node may make unavailable the node-specific resources (such as local storage and workloads scheduled to this node). Exercise caution when performing this operation to avoid impact on running services.

Step 4 Click **Yes**.

Step 5 Reconfigure node parameters.

If you need to adjust the container storage space, pay attention to the following configurations:

Storage Settings: Click **Expand** next to the data disk to set the following parameters:

Space Allocation for Pods: indicates the base size of a pod. It is the maximum size that a workload's pods (including the container images) can grow to in the disk space. Proper settings can prevent pods from taking all the disk space available and avoid service exceptions. It is recommended that the value is less than or equal to 80% of the container engine space. This parameter is related to the node OS and container storage rootfs and is not supported in some scenarios.

For more information about container storage space allocation, see **Data Disk Space Allocation**.

Step 6 After the node is reset, log in to the node and run the following command to access the container and check whether the container storage capacity has been expanded:

docker exec -it *container_id* **/bin/sh** or **kubectl exec -it** *container_id* **/bin/sh**

df -h

```
# df -h
Filesystem                                                                                                      Size  Used Avail Use% Mounted on
/dev/mapper/docker-253:1-787293-631c1bde2cbe82e39f32253b216ba914cb183b168b54700b3e5b9a54ee40a0d1   15G  229M   15G   2% /
tmpfs                                                                                                           32G     0   32G   0% /dev
tmpfs                                                                                                           32G     0   32G   0% /sys/fs/cgroup
/dev/mapper/vgpaas-kubernetes                                                                                  9.8G   37M  9.2G   1% /etc/hosts
/dev/vda1                                                                                                       40G  5.2G   33G  14% /etc/hostname
shm                                                                                                             64M     0   64M   0% /dev/shm
tmpfs                                                                                                           32G   16K   32G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                                                                                                           32G     0   32G   0% /proc/acpi
tmpfs                                                                                                           32G     0   32G   0% /sys/firmware
tmpfs                                                                                                           32G     0   32G   0% /proc/scsi
tmpfs                                                                                                           32G     0   32G   0% /proc/kbox
tmpfs                                                                                                           32G     0   32G   0% /proc/oom_extend
```

----End

Expanding a PVC

Cloud storage:

- OBS and SFS: There is no storage restriction and capacity expansion is not required.
- EVS:
  - You can expand the capacity of automatically created pay-per-use volumes on the console. The procedure is as follows:
    i. Choose **Storage** in the navigation pane. In the right pane, click the **PVCs** tab. Click **More** in the **Operation** column of the target PVC and select **Scale-out**.

ii. Enter the capacity to be added and click **OK**.

- For yearly/monthly-billed instances, expand the capacity on the EVS console and then change the capacity in the PVC.

- For SFS Turbo, expand the capacity on the SFS console and then change the capacity in the PVC.

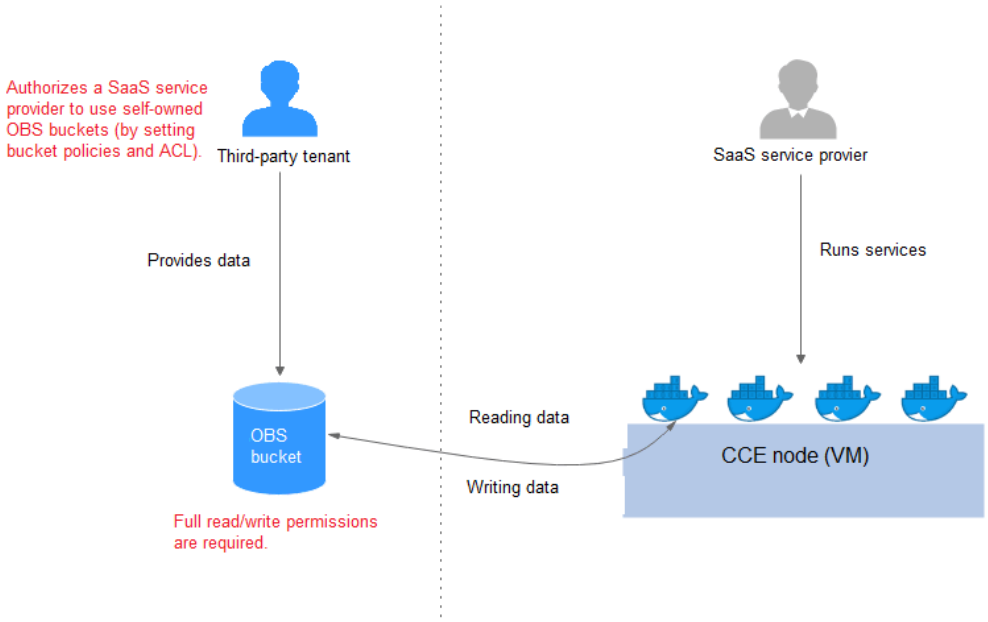
11.2 Mounting an Object Storage Bucket of a Third-Party Tenant

This section describes how to mount OBS buckets and OBS parallel file systems (preferred) of third-party tenants.

Application Scenarios

The CCE cluster of a SaaS service provider needs to be mounted with the OBS bucket of a third-party tenant, as shown in **Figure 11-1**.

Figure 11-1 Mounting an OBS bucket of a third-party tenant



1. **The third-party tenant authorizes the SaaS service provider to access the OBS buckets or parallel file systems** by setting the bucket policy and bucket ACL.
2. **The SaaS service provider statically imports the OBS buckets and parallel file systems of the third-party tenant**.
3. The SaaS service provider processes the service and writes the processing result (result file or result data) back to the OBS bucket of the third-party tenant.

Precautions

- Only parallel file systems and OBS buckets of third-party tenants in the same region can be mounted.
- Only clusters where the everest add-on of v1.1.11 or later has been installed (the cluster version must be v1.15 or later) can be mounted with OBS buckets of third-party tenants.
- The service platform of the SaaS service provider needs to manage the lifecycle of the third-party bucket PVs. When a PVC is deleted separately, the PV is not deleted. Instead, it will be retained. To do so, call the native Kubernetes APIs to create and delete static PVs.

Authorizing the SaaS Service Provider to Access the OBS Buckets

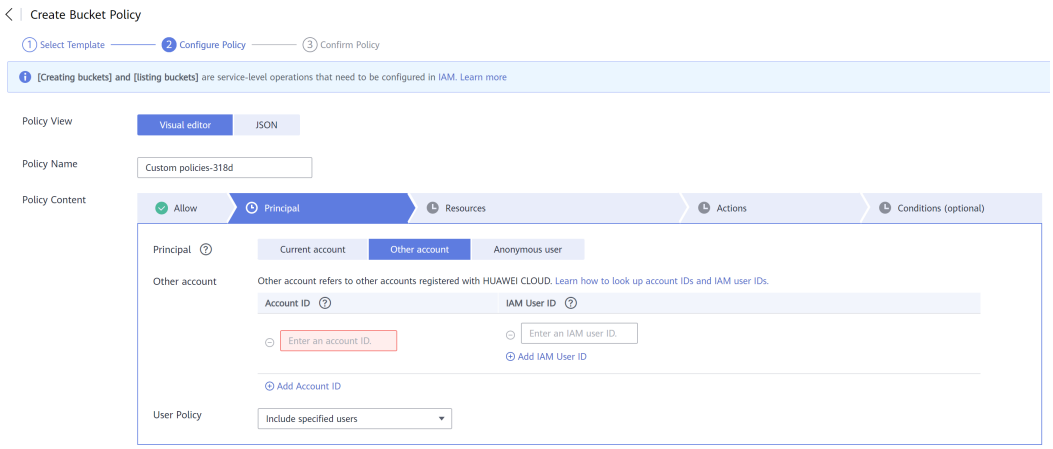
The following uses an OBS bucket as an example to describe how to set a bucket policy and bucket ACL to authorize the SaaS service provider. The configuration for an OBS parallel file system is the same.

Step 1 Log in to the OBS console.

Step 2 In the bucket list, click a bucket name and access the **Overview** page.

Step 3 In the navigation pane, choose **Permissions** > **Bucket Policies**. In the right pane, click **Create**. In the custom policy area, click **Create Custom Policy**.

Figure 11-2 Creating a bucket policy



- **Allow**: Select **Allow**.
- **Principal**: Select **Other account**, and enter the account ID and user ID. The bucket policy takes effect for the specified users.
- **Resources**: Select the resources that can be operated.
- **Actions**: Select the actions that can be operated.

Step 4 In the navigation pane, choose **Permissions** > **Bucket ACLs**. In the right pane, click **Add**. Enter the account ID of the authorized user, select **Read**, **Object read**, and **Write** for **Access to Bucket**, select **Read** and **Write** for **Access to ACL**, and click **OK**.

----End

Statically Importing OBS Buckets and Parallel File Systems

- **Static PV of an OBS bucket:**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: objbucket      #Replace the name with the actual PV name of the bucket.
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  mountOptions:
  - default_acl=bucket-owner-full-control      #New OBS mounting parameters
  csi:
    driver: obs.csi.everest.io
    fsType: s3fs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region: ap-southeast-1       #Set it to the ID of the current region.
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: objbucket            #Replace the name with the actual bucket name of the third-party tenant.
  persistentVolumeReclaimPolicy: Retain    #This parameter must be set to Retain to ensure that the bucket will not be deleted when a PV is deleted.
  storageClassName: csi-obs-mountoption    #You can associate a new custom OBS storage class or the built-in csi-obs of the cluster.
```

  - **mountOptions**: This field contains the new OBS mounting parameters that allow the bucket owner to have full access to the data in the bucket. This field solves the problem that the bucket owner cannot read the data written into a mounted third-party bucket. If the object storage of a third-party tenant is mounted, **default_acl** must be set to **bucket-owner-full-control**. For details about other values of **default_acl**, see **ACLs**.
  - **persistentVolumeReclaimPolicy**: When the object storage of a third-party tenant is mounted, this field must be set to **Retain**. In this way, the OBS bucket will not be deleted when a PV is deleted. The service platform of the SaaS service provider needs to manage the lifecycle of the third-party bucket PVs. When a PVC is deleted separately, the PV is not deleted. Instead, it will be retained. To do so, call the native Kubernetes APIs to create and delete static PVs.
  - **storageClassName**: You can associate a new custom OBS storage class (**click here**) or the built-in csi-obs of the cluster.

  PVC of a bound OBS bucket:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/fstype: obsfs
    everest.io/obs-volume-type: STANDARD
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: objbucketpvc      #Replace the name with the actual PVC name of the bucket.
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs-mountoption     #The value must be the same as the storage class associated with the bound PV.
  volumeName: objbucket       #Replace the name with the actual PV name of the bucket to be bound.
```


- **Static PV of an OBS parallel file system:**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: obsfscheck   #Replace the name with the actual PV name of the parallel file system.
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  mountOptions:
  - default_acl=bucket-owner-full-control     #New OBS mounting parameters
  csi:
    driver: obs.csi.everest.io
    fsType: obsfs
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD
      everest.io/region:  ap-southeast-1
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: obsfscheck              #Replace the name with the actual name of the parallel file system of the third-party tenant.
  persistentVolumeReclaimPolicy: Retain       #This parameter must be set to Retain to ensure that the bucket will not be deleted when a PV is deleted.
  storageClassName: csi-obs-mountoption       #You can associate a new custom OBS storage class or the built-in csi-obs of the cluster.
```

  - **mountOptions**: This field contains the new OBS mounting parameters that allow the bucket owner to have full access to the data in the bucket. This field solves the problem that the bucket owner cannot read the data written into a mounted third-party bucket. If the object storage of a third-party tenant is mounted, **default_acl** must be set to **bucket-owner-full-control**. For details about other values of **default_acl**, see **ACLs**.
  - **persistentVolumeReclaimPolicy**: When the object storage of a third-party tenant is mounted, this field must be set to **Retain**. In this way, the OBS bucket will not be deleted when a PV is deleted. The service platform of the SaaS service provider needs to manage the lifecycle of the third-party bucket PVs. When a PVC is deleted separately, the PV is not deleted. Instead, it will be retained. To do so, call the native Kubernetes APIs to create and delete static PVs.
  - **storageClassName**: You can associate a new custom OBS storage class (**click here**) or the built-in csi-obs of the cluster.

  PVC of a bound OBS parallel file system:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    csi.storage.k8s.io/fstype: obsfs
    everest.io/obs-volume-type: STANDARD
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: obsfscheckpvc   #Replace the name with the actual PVC name of the parallel file system.
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs-mountoption    #The value must be the same as the storage class associated with the bound PV.
  volumeName: obsfscheck     #Replace the name with the actual PV name of the parallel file system.
```


- **(Optional) Creating a custom OBS storage class to associate with a static PV:**

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-obs-mountoption
mountOptions:
  - default_acl=bucket-owner-full-control
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: obsfs
  everest.io/obs-volume-type: STANDARD
provisioner: everest-csi-provisioner
reclaimPolicy: Retain
volumeBindingMode: Immediate
```

  - **csi.storage.k8s.io/fstype**: File type. The value can be **obsfs** or **s3fs**. If the value is **s3fs**, an OBS bucket is created and mounted using s3fs. If the value is **obsfs**, an OBS parallel file system is created and mounted using obsfs.
  - **reclaimPolicy**: Reclaim policy of a PV. The value will be set in **PV.spec.persistentVolumeReclaimPolicy** dynamically created based on the new PVC associated with the storage class. If the value is **Delete**, the external OBS bucket and the PV will be deleted when the PVC is deleted. If the value is **Retain**, the PV and external storage are retained when the PVC is deleted. In this case, clear the PV separately. In the scenario where an imported third-party bucket is associated, the storage class is used only for associating static PVs (with this field set to **Retain**). Dynamic creation is not involved.

11.3 Using StorageClass to Dynamically Create a Subdirectory in an SFS Turbo File System

Background

The minimum capacity of an SFS Turbo file system is 500 GiB, and the SFS Turbo file system cannot be billed by usage. By default, the root directory of an SFS Turbo file system is mounted to a container which, in most case, does not require such a large capacity.

The everest add-on allows you to dynamically create subdirectories in an SFS Turbo file system and mount these subdirectories to containers. In this way, an SFS Turbo file system can be shared by multiple containers to increase storage efficiency.

Constraints

- Only clusters of v1.15 or later are supported.
- The cluster must use the everest add-on of version 1.1.13 or later.
- Kata containers are not supported.
- When the everest add-on earlier than 1.2.69 or 2.1.11 is used, a maximum of 10 PVCs can be created concurrently at a time by using the subdirectory function. everest of 1.2.69 or later or of 2.1.11 or later is recommended.
- A subPath volume is a subdirectory of an SFS Turbo file system. Increasing the capacity of a PVC of this type only changes the resource range specified by

the PVC, but does not change the total capacity of the SFS Turbo file system. If the SFS Turbo file system's total resource capacity is not enough, the available capacity of the subPath volume will be restricted. To fix this, you must increase the resource capacity of the SFS Turbo file system on the SFS Turbo console.

Deleting the subPath volume does not result in the deletion of the resources of the SFS Turbo file system.

Creating an SFS Turbo Volume of the subPath Type

Step 1 Create an SFS Turbo file system in the same VPC and subnet as the cluster.

Step 2 Create a YAML file of StorageClass, for example, **sfsturbo-subpath-sc.yaml**.

The following is an example:

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  name: sfsturbo-subpath-sc
mountOptions:
- lock
parameters:
  csi.storage.k8s.io/csi-driver-name: sfsturbo.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/archive-on-delete: "true"
  everest.io/share-access-to: 7ca2dba2-1234-1234-1234-626371a8fb3a
  everest.io/share-expand-type: bandwidth
  everest.io/share-export-location: 192.168.1.1:/sfsturbo/
  everest.io/share-source: sfs-turbo
  everest.io/share-volume-type: STANDARD
  everest.io/volume-as: subpath
  everest.io/volume-id: 0d773f2e-1234-1234-1234-de6a35074696
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

In this example:

- **name**: indicates the name of the StorageClass.
- **mountOptions**: indicates the mount options. This field is optional.
  - In versions later than everest 1.1.13 and earlier than everest 1.2.8, only the **nolock** parameter can be configured. By default, the **nolock** parameter is used for the mount operation and does not need to be configured. If **nolock** is set to **false**, the **lock** field is used.
  - Starting from everest 1.2.8, more mount options are supported. For details, see **Configuring SFS Turbo Mount Options**. **Do not set nolock to true. Otherwise, the mount operation will fail.**

    ```
    mountOptions:
    - vers=3
    - timeo=600
    - nolock
    - hard
    ```

- **everest.io/volume-as**: This parameter is set to **subpath** to use the subPath volume.
- **everest.io/share-access-to**: This parameter is optional. In a subPath volume, set this parameter to the ID of the VPC where the SFS Turbo file system is located.

- **everest.io/share-expand-type**: This parameter is optional. If the type of the SFS Turbo file system is SFS Turbo Standard – Enhanced or SFS Turbo Performance – Enhanced, set this parameter to **bandwidth**.
- **everest.io/share-export-location**: This parameter indicates the mount directory. It consists of the SFS Turbo shared path and sub-directory. The shared path can be obtained on the SFS Turbo console. The sub-directory is user-defined. The PVCs created using the StorageClass are located in this sub-directory.
- **everest.io/share-volume-type**: This parameter is optional. It specifies the SFS Turbo file system type. The value can be **STANDARD** or **PERFORMANCE**. For enhanced types, this parameter must be used together with **everest.io/share-expand-type** (whose value should be **bandwidth**).
- **everest.io/zone**: This parameter is optional. Set it to the AZ where the SFS Turbo file system is located.
- **everest.io/volume-id**: This parameter indicates the ID of the SFS Turbo volume. You can obtain the volume ID on the SFS Turbo page.
- **everest.io/archive-on-delete**: If this parameter is set to **true** and **Delete** is selected for **Reclaim Policy**, the original documents of the PV will be archived to the directory named **archived-***{$PV name.timestamp}* before the PVC is deleted. If this parameter is set to **false**, the SFS Turbo subdirectory of the corresponding PV will be deleted. The default value is **true**, indicating that the original documents of the PV will be archived to the directory named **archived-***{$PV name.timestamp}* before the PVC is deleted.

Step 3 Run **kubectl create -f sfsturbo-subpath-sc.yaml**.

Step 4 Create a PVC YAML file named **sfs-turbo-test.yaml**.

The following is an example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sfs-turbo-test
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
  storageClassName: sfsturbo-subpath-sc
  volumeMode: Filesystem
```

In this example:

- **name**: indicates the name of the PVC.
- **storageClassName**: specifies the name of the StorageClass.
- **storage**: In a subPath volume, modifying the value of this parameter does not impact the resource capacity of the SFS Turbo file system. A subPath volume is essentially a file path within an SFS Turbo file system. As a result, increasing the capacity of the subPath volume in a PVC does not lead to an increase in the resources of the SFS Turbo file system.

> **NOTE**
>
> The capacity of a subPath volume is restricted by the overall resource capacity of the corresponding SFS Turbo file system. If the resources of the SFS Turbo file system are inadequate, you can adjust the resource capacity via the SFS Turbo console.

Step 5 Run **kubectl create -f sfs-turbo-test.yaml**.

----End

Creating a Deployment and Mounting an Existing Volume

Step 1 Create a YAML file for the Deployment, for example, **deployment-test.yaml**.

The following is an example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-turbo-subpath-example
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-turbo-subpath-example
  template:
    metadata:
      labels:
        app: test-turbo-subpath-example
    spec:
      containers:
      - image: nginx:latest
        name: container-0
        volumeMounts:
        - mountPath: /tmp
          name: pvc-sfs-turbo-example
      restartPolicy: Always
      imagePullSecrets:
      - name: default-secret
      volumes:
      - name: pvc-sfs-turbo-example
        persistentVolumeClaim:
          claimName: sfs-turbo-test
```

In this example:

- **name**: indicates the name of the created workload.
- **image**: specifies the image used by the workload.
- **mountPath**: indicates the mount path of the container. In this example, the volume is mounted to the **/tmp** directory.
- **claimName**: indicates the name of an existing PVC.

Step 2 Create the Deployment.

kubectl create -f deployment-test.yaml

----End

Dynamically Creating a subPath Volume for a StatefulSet

Step 1 Create a YAML file for a StatefulSet, for example, **statefulset-test.yaml**.

The following is an example:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: test-turbo-subpath
  namespace: default
  generation: 1
  labels:
    appgroup: ''
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test-turbo-subpath
  template:
    metadata:
      labels:
        app: test-turbo-subpath
      annotations:
        metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
        pod.alpha.kubernetes.io/initialized: 'true'
    spec:
      containers:
        - name: container-0
          image: 'nginx:latest'
          resources: {}
          volumeMounts:
            - name: sfs-turbo-160024548582479676
              mountPath: /tmp
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: IfNotPresent
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: default-secret
      affinity: {}
      schedulerName: default-scheduler
  volumeClaimTemplates:
    - metadata:
        name: sfs-turbo-160024548582479676
        namespace: default
        annotations: {}
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 10Gi
        storageClassName: sfsturbo-subpath-sc
  serviceName: wwww
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
  revisionHistoryLimit: 10
```

In this example:

- **name**: indicates the name of the created workload.
- **image**: specifies the image used by the workload.

- **mountPath**: indicates the mount path of the container. In this example, the volume is mounted to the **/tmp** directory.
- **spec.template.spec.containers.volumeMounts.name** and **spec.volumeClaimTemplates.metadata.name**: must be consistent because they have a mapping relationship.
- **storageClassName**: specifies the name of an on-premises StorageClass.

Step 2 Create the StatefulSet.

kubectl create -f statefulset-test.yaml

----End

11.4 How Do I Change the Storage Class Used by a Cluster of v1.15 from FlexVolume to CSI Everest?

In clusters later than v1.15.11-r1, CSI (the everest add-on) has taken over all functions of fuxi FlexVolume (the storage-driver add-on) for managing container storage. You are advised to use CSI Everest.

To migrate your storage volumes, create a static PV to associate with the original underlying storage, and then create a PVC to associate with this static PV. When you upgrade your application, mount the new PVC to the original mounting path to migrate the storage volumes.

WARNING

Services will be interrupted during the migration. Therefore, properly plan the migration and back up data.

Procedure

Step 1 (Optional) Back up data to prevent data loss in case of exceptions.

Step 2 Configure a YAML file of the PV in the CSI format according to the PV in the FlexVolume format and associate the PV with the existing storage.

To be specific, run the following commands to configure the pv-example.yaml file, which is used to create a PV.

touch pv-example.yaml

vi pv-example.yaml

Configuration example of **a PV for an EVS volume**:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    failure-domain.beta.kubernetes.io/region: ap-southeast-1
    failure-domain.beta.kubernetes.io/zone: <zone name>
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
  name: pv-evs-example
```



```
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 10Gi
  csi:
    driver: disk.csi.everest.io
    fsType: ext4
    volumeAttributes:
      everest.io/disk-mode: SCSI
      everest.io/disk-volume-type: SAS
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 0992dbda-6340-470e-a74e-4f0db288ed82
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-disk
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-2 EVS volume configuration parameters

Parameter	Description
failure-domain.beta.kubernetes.io/region	Region where the EVS disk is located. Use the same value as that of the FlexVolume PV.
failure-domain.beta.kubernetes.io/zone	AZ where the EVS disk is located. Use the same value as that of the FlexVolume PV.
name	Name of the PV, which must be unique in the cluster.
storage	EVS volume capacity in the unit of Gi. Use the value of **spec.capacity.storage** of the FlexVolume PV.
driver	Storage driver used to attach the volume. Set the driver to **disk.csi.everest.io** for the EVS volume.
volumeHandle	Volume ID of the EVS disk. Use the value of **spec.flexVolume.options.volumeID** of the FlexVolume PV.
everest.io/disk-mode	EVS disk mode. Use the value of **spec.flexVolume.options.disk-mode** of the FlexVolume PV.
everest.io/disk-volume-type	EVS disk type. Currently, high I/O (SAS) and ultra-high I/O (SSD) are supported. Use the value of **kubernetes.io/volumetype** in the storage class corresponding to **spec.storageClassName** of the FlexVolume PV.
storageClassName	Name of the Kubernetes storage class associated with the storage volume. Set this field to **csi-disk** for EVS disks.

Configuration example of **a PV for an SFS volume**:

```
apiVersion: v1
kind: PersistentVolume
```



```
metadata:
  name: pv-sfs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: nas.csi.everest.io
    fsType: nfs
    volumeAttributes:
      everest.io/share-export-location: sfs-nas01.ap-southeast-1.myhuaweicloud.com:/share-436304e8
      storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    volumeHandle: 682f00bb-ace0-41d8-9b3e-913c9aa6b695
  persistentVolumeReclaimPolicy: Delete
  storageClassName: csi-nas
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-3 SFS volume configuration parameters

Parameter	Description
name	Name of the PV, which must be unique in the cluster.
storage	File storage size in the unit of Gi. Use the value of **spec.capacity.storage** of the FlexVolume PV.
driver	Storage driver used to attach the volume. Set the driver to **nas.csi.everest.io** for the file system.
everest.io/share-export-location	Shared path of the file system. Use the value of **spec.flexVolume.options.deviceMountPath** of the FlexVolume PV.
volumeHandle	File system ID. Use the value of **spec.flexVolume.options.volumeID** of the FlexVolume PV.
storageClassName	Name of the Kubernetes storage class. Set this field to **csi-nas**.

Configuration example of **a PV for an OBS volume**:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-obs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 1Gi
  csi:
    driver: obs.csi.everest.io
    fsType: s3fs
    volumeAttributes:
```



```
    everest.io/obs-volume-type: STANDARD
    everest.io/region: ap-southeast-1
    storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
  volumeHandle: obs-normal-static-pv
persistentVolumeReclaimPolicy: Delete
storageClassName: csi-obs
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-4 OBS volume configuration parameters

Parameter	Description
name	Name of the PV, which must be unique in the cluster.
storage	Storage capacity, in the unit of Gi. Set this parameter to the fixed value **1Gi**.
driver	Storage driver used to attach the volume. Set the driver to **obs.csi.everest.io** for the OBS volume.
fsType	File type. Value options are **obsfs** or **s3fs**. If the value is **s3fs**, an OBS bucket is created and mounted using s3fs. If the value is **obsfs**, an OBS parallel file system is created and mounted using obsfs. Set this parameter according to the value of **spec.flexVolume.options.posix** of the FlexVolume PV. If the value of **spec.flexVolume.options.posix** is **true**, set this parameter to **obsfs**. If the value is **false**, set this parameter to **s3fs**.
everest.io/obs-volume-type	Storage class, including **STANDARD** (standard bucket) and **WARM** (infrequent access bucket). Set this parameter according to the value of **spec.flexVolume.options.storage_class** of the FlexVolume PV. If the value of **spec.flexVolume.options.storage_class** is **standard**, set this parameter to **STANDARD**. If the value is **standard_ia**, set this parameter to **WARM**.
everest.io/region	Region where the OBS bucket is located. Use the value of **spec.flexVolume.options.region** of the FlexVolume PV.
volumeHandle	OBS bucket name. Use the value of **spec.flexVolume.options.volumeID** of the FlexVolume PV.
storageClassName	Name of the Kubernetes storage class. Set this field to **csi-obs**.

Configuration example of **a PV for an SFS Turbo volume**:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-efs-example
  annotations:
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
```



```
accessModes:
- ReadWriteMany
capacity:
  storage: 10Gi
csi:
  driver: sfsturbo.csi.everest.io
  fsType: nfs
  volumeAttributes:
    everest.io/share-export-location: 192.168.0.169:/
    storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
  volumeHandle: 8962a2a2-a583-4b7f-bb74-fe76712d8414
persistentVolumeReclaimPolicy: Delete
storageClassName: csi-sfsturbo
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-5 SFS Turbo volume configuration parameters

Parameter	Description
name	Name of the PV, which must be unique in the cluster.
storage	File system size. Use the value of **spec.capacity.storage** of the FlexVolume PV.
driver	Storage driver used to attach the volume. Set it to **sfsturbo.csi.everest.io**.
everest.io/share-export-location	Shared path of the SFS Turbo volume. Use the value of **spec.flexVolume.options.deviceMountPath** of the FlexVolume PV.
volumeHandle	SFS Turbo volume ID. Use the value of **spec.flexVolume.options.volumeID** of the FlexVolume PV.
storageClassName	Name of the Kubernetes storage class. Set this field to **csi-sfsturbo** for SFS Turbo volumes.

Step 3 Configure a YAML file of the PVC in the CSI format according to the PVC in the FlexVolume format and associate the PVC with the PV created in **Step 2**.

To be specific, run the following commands to configure the pvc-example.yaml file, which is used to create a PVC.

touch pvc-example.yaml

vi pvc-example.yaml

Configuration example of **a PVC for an EVS volume**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    failure-domain.beta.kubernetes.io/region: ap-southeast-1
    failure-domain.beta.kubernetes.io/zone: <zone name>
  annotations:
    everest.io/disk-volume-type: SAS
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-evs-example
```



```
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeName:  pv-evs-example
  storageClassName: csi-disk
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-6 PVC configuration parameters for an EVS volume

Parameter	Description
failure-domain.beta.kubernetes.io/region	Region where the cluster is located. Use the same value as that of the FlexVolume PVC.
failure-domain.beta.kubernetes.io/zone	AZ where the EVS disk is deployed. Use the same value as that of the FlexVolume PVC.
everest.io/disk-volume-type	Storage class of the EVS disk. The value can be **SAS** or **SSD**. Set this parameter to the same value as that of the PV created in **Step 2**.
name	PVC name, which must be unique in the namespace. The value must be unique in the namespace. (If the PVC is dynamically created by a stateful application, the value of this parameter must be the same as the name of the FlexVolume PVC.)
namespace	Namespace to which the PVC belongs. Use the same value as that of the FlexVolume PVC.
storage	Requested capacity of the PVC, which must be the same as the storage size of the existing PV.
volumeName	Name of the PV. Set this parameter to the name of the static PV in **Step 2**.
storageClassName	Name of the Kubernetes storage class. Set this field to **csi-disk** for EVS disks.

Configuration example of **a PVC for an SFS volume**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-sfs-example
  namespace: default
spec:
  accessModes:
```



```
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-nas
  volumeName: pv-sfs-example
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-7 PVC configuration parameters for an SFS volume

Parameter	Description
name	PVC name, which must be unique in the namespace. The value must be unique in the namespace. (If the PVC is dynamically created by a stateful application, the value of this parameter must be the same as the name of the FlexVolume PVC.)
namespace	Namespace to which the PVC belongs. Use the same value as that of the FlexVolume PVC.
storage	Storage capacity, in the unit of Gi. The value must be the same as the storage size of the existing PV.
storageClassName	Set this field to **csi-nas**.
volumeName	Name of the PV. Set this parameter to the name of the static PV in **Step 2**.

Configuration example of **a PVC for an OBS volume**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
    everest.io/obs-volume-type: STANDARD
    csi.storage.k8s.io/fstype: s3fs
  name: pvc-obs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: csi-obs
  volumeName: pv-obs-example
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-8 PVC configuration parameters for an OBS volume

Parameter	Description
everest.io/obs-volume-type	OBS volume type, which can be **STANDARD** (standard bucket) and **WARM** (infrequent access bucket). Set this parameter to the same value as that of the PV created in **Step 2**.
csi.storage.k8s.io/fstype	File type, which can be **obsfs** or **s3fs**. The value must be the same as that of **fsType** of the static OBS volume PV.
name	PVC name, which must be unique in the namespace. The value must be unique in the namespace. (If the PVC is dynamically created by a stateful application, the value of this parameter must be the same as the name of the FlexVolume PVC.)
namespace	Namespace to which the PVC belongs. Use the same value as that of the FlexVolume PVC.
storage	Storage capacity, in the unit of Gi. Set this parameter to the fixed value **1Gi**.
storageClassName	Name of the Kubernetes storage class. Set this field to **csi-obs**.
volumeName	Name of the PV. Set this parameter to the name of the static PV created in **Step 2**.

Configuration example of **a PVC for an SFS Turbo volume**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
  name: pvc-efs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-sfsturbo
  volumeName: pv-efs-example
```

Pay attention to the fields in bold and red. The parameters are described as follows:

Table 11-9 PVC configuration parameters for an SFS Turbo volume

Parameter	Description
name	PVC name, which must be unique in the namespace. The value must be unique in the namespace. (If the PVC is dynamically created by a stateful application, the value of this parameter must be the same as the name of the FlexVolume PVC.)
namespace	Namespace to which the PVC belongs. Use the same value as that of the FlexVolume PVC.
storageClassName	Name of the Kubernetes storage class. Set this field to **csi-sfsturbo**.
storage	Storage capacity, in the unit of Gi. The value must be the same as the storage size of the existing PV.
volumeName	Name of the PV. Set this parameter to the name of the static PV created in **Step 2**.

Step 4 Upgrade the workload to use a new PVC.

For Deployments

1. Run the **kubectl create -f** commands to create a PV and PVC.

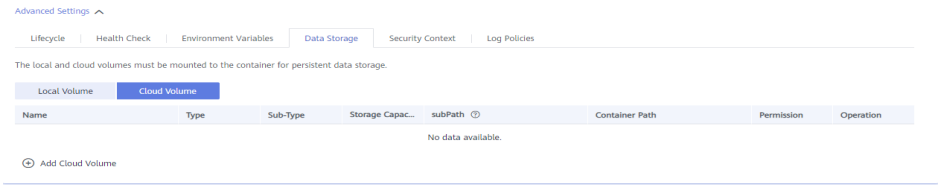
   kubectl create -f pv-example.yaml

   kubectl create -f pvc-example.yaml

   NOTE

   Replace the example file name **pvc-example.yaml** in the preceding commands with the names of the YAML files configured in **Step 2** and **Step 3**.

2. Go to the CCE console. On the workload upgrade page, click **Upgrade** > **Advanced Settings** > **Data Storage** > **Cloud Storage**.



3. Uninstall the old storage and add the PVC in the CSI format. Retain the original mounting path in the container.
4. Click **Submit**.
5. Wait until the pods are running.

For StatefulSets that use existing storage

1. Run the **kubectl create -f** commands to create a PV and PVC.

   kubectl create -f pv-example.yaml

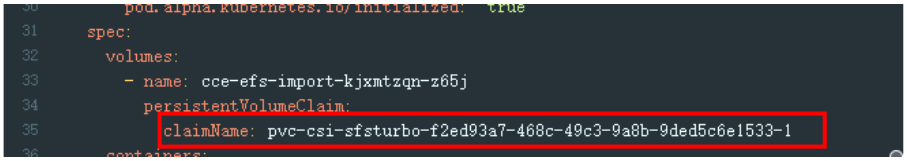
   kubectl create -f pvc-example.yaml

NOTE

Replace the example file name **pvc-example.yaml** in the preceding commands with the names of the YAML files configured in **Step 2** and **Step 3**.

2. Run the **kubectl edit** command to edit the StatefulSet and use the newly created PVC.

   kubectl edit sts sts-example -n xxx



NOTE

Replace **sts-example** in the preceding command with the actual name of the StatefulSet to upgrade. **xxx** indicates the namespace to which the StatefulSet belongs.

3. Wait until the pods are running.

NOTE

The current console does not support the operation of adding new cloud storage for StatefulSets. Use the kubectl commands to replace the storage with the newly created PVC.

For StatefulSets that use dynamically allocated storage

1. Back up the PV and PVC in the flexVolume format used by the StatefulSet.

   kubectl get pvc xxx -n {namespaces} -oyaml > pvc-backup.yaml

   kubectl get pv xxx -n {namespaces} -oyaml > pv-backup.yaml

2. Change the number of pods to **0**.
3. On the storage page, disassociate the flexVolume PVC used by the StatefulSet.
4. Run the **kubectl create -f** commands to create a PV and PVC.

   kubectl create -f pv-example.yaml

   kubectl create -f pvc-example.yaml

   NOTE

   Replace the example file name **pvc-example.yaml** in the preceding commands with the names of the YAML files configured in **Step 2** and **Step 3**.

5. Change the number of pods back to the original value and wait until the pods are running.

NOTE

The dynamic allocation of storage for StatefulSets is achieved by using **volumeClaimTemplates**. This field cannot be modified by Kubernetes. Therefore, data cannot be migrated by using a new PVC.

The PVC naming rule of the **volumeClaimTemplates** is fixed. When a PVC that meets the naming rule exists, this PVC is used.

Therefore, disassociate the original PVC first, and then create a PVC with the same name in the CSI format.

6. (Optional) Recreate the stateful application to ensure that a CSI PVC is used when the application is scaled out. Otherwise, FlexVolume PVCs are used in scaling out.

- Run the following command to obtain the YAML file of the StatefulSet:

kubectl get sts xxx -n {namespaces} -oyaml > sts.yaml

- Run the following command to back up the YAML file of the StatefulSet:

cp sts.yaml sts-backup.yaml

- Modify the definition of **volumeClaimTemplates** in the YAML file of the StatefulSet.

vi sts.yaml

Configuration example of **volumeClaimTemplates for an EVS volume**:

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161070049798261342
      namespace: default
      creationTimestamp: null
      annotations:
        everest.io/disk-volume-type: SAS
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-disk
```

The parameter value must be the same as the PVC of the EVS volume created in **Step 3**.

Configuration example of **volumeClaimTemplates for an SFS volume**:

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161063441560279697
      namespace: default
      creationTimestamp: null
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-nas
```

The parameter value must be the same as the PVC of the SFS volume created in **Step 3**.

Configuration example of **volumeClaimTemplates for an OBS volume**:

```
volumeClaimTemplates:
  - metadata:
      name: pvc-161070100417416148
      namespace: default
      creationTimestamp: null
      annotations:
        csi.storage.k8s.io/fstype: s3fs
        everest.io/obs-volume-type: STANDARD
    spec:
      accessModes:
        - ReadWriteMany
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-obs
```


The parameter value must be the same as the PVC of the OBS volume created in **Step 3**.

- Delete the StatefulSet.

kubectl delete sts xxx -n {namespaces}

- Create the StatefulSet.

kubectl create -f sts.yaml

Step 5 Check service functions.

1. Check whether the application is running properly.
2. Checking whether the data storage is normal.

NOTE

If a rollback is required, perform **Step 4**. Select the PVC in FlexVolume format and upgrade the application.

Step 6 Uninstall the PVC in the FlexVolume format.

If the application functions normally, unbind the PVC in the FlexVolume format on the storage management page.

You can also run the kubectl command to delete the PVC and PV of the FlexVolume format.

CAUTION

Before deleting a PV, change the persistentVolumeReclaimPolicy of the PV to **Retain**. Otherwise, the underlying storage will be reclaimed after the PV is deleted.

If the cluster has been upgraded before the storage migration, PVs may fail to be deleted. You can remove the PV protection field **finalizers** to delete PVs.

kubectl patch pv {pv_name} -p '{"metadata":{"finalizers":null}}'

----End

11.5 Custom Storage Classes

Background

When using storage resources in CCE, the most common method is to specify **storageClassName** to define the type of storage resources to be created when creating a PVC. The following configuration shows how to use a PVC to apply for an SAS (high I/O) EVS disk (block storage).

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS
```



```
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
```

To specify the EVS disk type, you can configure the **everest.io/disk-volume-type** field. The value **SAS** is used as an example here, indicating the high I/O EVS disk type. Or you can choose **SSD** (ultra-high I/O).

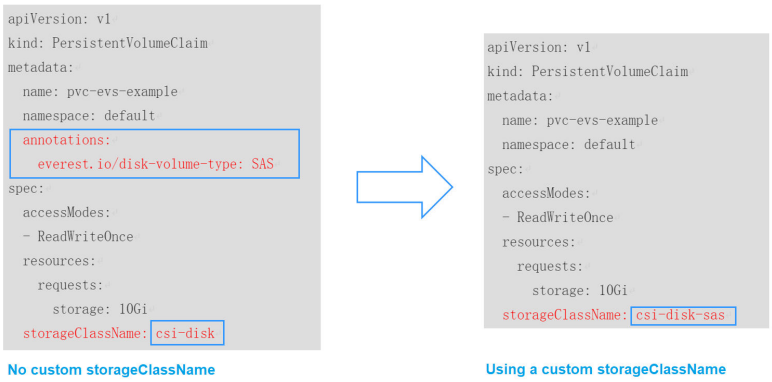
This configuration method may not work if you want to:

- Set **storageClassName** only, which is simpler than specifying the EVS disk type by using **everest.io/disk-volume-type**.
- Avoid modifying YAML files or Helm charts. Some users switch from self-built or other Kubernetes services to CCE and have written YAML files of many applications. In these YAML files, different types of storage resources are specified by different StorageClassNames. When using CCE, they need to modify a large number of YAML files or Helm charts to use storage resources, which is labor-consuming and error-prone.
- Set the default **storageClassName** for all applications to use the default storage class. In this way, you can create storage resources of the default type without needing to specify **storageClassName** in the YAML file.

Solution

This section describes how to set a custom storage class in CCE and how to set the default storage class. You can specify different types of storage resources by setting **storageClassName**.

- For the first scenario, you can define custom storageClassNames for SAS and SSD EVS disks. For example, define a storage class named **csi-disk-sas** for creating SAS disks. The following figure shows the differences before and after you use a custom storage class.



- For the second scenario, you can define a storage class with the same name as that in the existing YAML file without needing to modify **storageClassName** in the YAML file.
- For the third scenario, you can set the default storage class as described below to create storage resources without specifying **storageClassName** in YAML files.

```
apiVersion: v1
kind: PersistentVolumeClaim
```



```
metadata:
  name: pvc-evs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

CCE Default Storage Classes

As of now, CCE provides storage classes such as csi-disk, csi-nas, and csi-obs by default. When defining a PVC, you can use a **storageClassName** to automatically create a PV of the corresponding type and automatically create underlying storage resources.

Run the following kubectl command to obtain the storage classes that CCE supports. Use the CSI add-on provided by CCE to create a storage class.

```
# kubectl get sc
NAME                PROVISIONER                     AGE
csi-disk            everest-csi-provisioner         17d         # EVS disk
csi-disk-topology   everest-csi-provisioner         17d         # EVS disks created with delay
csi-nas             everest-csi-provisioner         17d         # SFS 1.0
csi-obs             everest-csi-provisioner         17d         # OBS
csi-sfsturbo        everest-csi-provisioner         17d         # SFS Turbo

csi-local           everest-csi-provisioner         17d         # Local PV

csi-local-topology  everest-csi-provisioner         17d         # Local PV created with delay
```

Each storage class contains the default parameters used for dynamically creating a PV. The following is an example of storage class for EVS disks:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

Table 11-10 Key parameters

Parameter	Description
provisioner	Specifies the storage resource provider, which is the Everest add-on for CCE. Set this parameter to **everest-csi-provisioner**.
parameters	Specifies the storage parameters, which vary with storage types. For details, see **Table 11-11**.

Parameter	Description
reclaimPolicy	Specifies the value of **persistentVolumeReclaimPolicy** for creating a PV. The value can be **Delete** or **Retain**. If **reclaimPolicy** is not specified when a StorageClass object is created, the value defaults to **Delete**. • **Delete**: indicates that a dynamically created PV will be automatically deleted when the PVC is deleted. • **Retain**: indicates that a dynamically created PV will be retained when the PVC is deleted.
allowVolumeExpansion	Specifies whether the PV of this storage class supports dynamic capacity expansion. The default value is **false**. Dynamic capacity expansion is implemented by the underlying storage add-on. This is only a switch.
volumeBindingMode	Specifies the volume binding mode, that is, the time when a PV is dynamically created. The value can be **Immediate** or **WaitForFirstConsumer**. • **Immediate**: PV binding and dynamic creation are completed when a PVC is created. • **WaitForFirstConsumer**: PV binding and creation are delayed. The PV creation and binding processes are executed only when the PVC is used in the workload.
mountOptions	This field must be supported by the underlying storage. If this field is not supported but is specified, the PV creation will fail.

Table 11-11 Parameters

Volume Type	Parameter	Mandatory	Description
EVS	csi.storage.k8s.io/csi-driver-name	Yes	Driver type. If an EVS disk is used, the parameter value is fixed at **disk.csi.everest.io**.
	csi.storage.k8s.io/fstype	Yes	If an EVS disk is used, the parameter value can be **ext4**.

Volume Type	Parameter	Mandatory	Description
	everest.io/disk-volume-type	Yes	EVS disk type. All letters are in uppercase. • **SAS**: high I/O • **SSD**: ultra-high I/O • **GPSSD**: general-purpose SSD • **ESSD**: extreme SSD • **GPSSD2**: general-purpose SSD v2, which is supported when the Everest version is 2.4.4 or later and the **everest.io/disk-iops** and **everest.io/disk-throughput** annotations are configured
	everest.io/passthrough	Yes	The parameter value is fixed at **true**, which indicates that the EVS device type is **SCSI**. Other parameter values are not allowed.
SFS	csi.storage.k8s.io/csi-driver-name	Yes	Driver type. If SFS is used, the parameter value is fixed at **nas.csi.everest.io**.
	csi.storage.k8s.io/fstype	Yes	If SFS is used, the value can be **nfs**.
	everest.io/share-access-level	Yes	The parameter value is fixed at **rw**, indicating that the SFS data is readable and writable.
	everest.io/share-access-to	Yes	VPC ID of the cluster.
	everest.io/share-is-public	No	The parameter value is fixed at **false**, indicating that the file is shared to private. You do not need to configure this parameter when SFS 3.0 is used.
	everest.io/sfs-version	No	This parameter is mandatory only when SFS 3.0 is used. The value is fixed at **sfs3.0**.
SFS Turbo	csi.storage.k8s.io/csi-driver-name	Yes	Driver type. If SFS Turbo is used, the parameter value is fixed at **sfsturbo.csi.everest.io**.
	csi.storage.k8s.io/fstype	Yes	If SFS Turbo is used, the value can be **nfs**.

Volume Type	Parameter	Mandatory	Description
	everest.io/share-access-to	Yes	VPC ID of the cluster.
	everest.io/share-expand-type	No	Extension type. The default value is **bandwidth**, indicating an enhanced file system. This parameter does not take effect.
	everest.io/share-source	Yes	The parameter value is fixed at **sfs-turbo**.
	everest.io/share-volume-type	No	SFS Turbo storage class. The default value is **STANDARD**, indicating standard and standard enhanced editions. This parameter does not take effect.
OBS	csi.storage.k8s.io/csi-driver-name	Yes	Driver type. If OBS is used, the parameter value is fixed at **obs.csi.everest.io**.
	csi.storage.k8s.io/fstype	Yes	Instance type, which can be **obsfs** or **s3fs**. • **obsfs**: Parallel file system, which is mounted using obsfs (recommended). • **s3fs**: Object bucket, which is mounted using s3fs.
	everest.io/obs-volume-type	Yes	OBS storage class. • If **fsType** is set to **s3fs**, **STANDARD** (standard bucket) and **WARM** (infrequent access bucket) are supported. • This parameter is invalid when **fsType** is set to **obsfs**.

Custom Storage Classes

You can customize a high I/O storage class in a YAML file. For example, the name **csi-disk-sas** indicates that the disk type is SAS (high I/O).

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-sas                      # Name of the high I/O storage class, which can be customized.
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS            # High I/O EVS disk type, which cannot be customized.
```



```
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true                  # true indicates that capacity expansion is allowed.
```

For an ultra-high I/O storage class, you can set the class name to **csi-disk-ssd** to create SSD EVS disk (ultra-high I/O).

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd                        # Name of the ultra-high I/O storage class, which can be customized.
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD          # Ultra-high I/O EVS disk type, which cannot be customized.
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

reclaimPolicy: indicates the recycling policies of the underlying cloud storage. The value can be **Delete** or **Retain**.

- **Delete**: When a PVC is deleted, both the PV and the EVS disk are deleted.
- **Retain**: When a PVC is deleted, the PV and underlying storage resources are not deleted. Instead, you must manually delete these resources. After that, the PV resource is in the **Released** state and cannot be bound to the PVC again.

NOTE

The reclamation policy configured here has no impact on the SFS Turbo storage and the subscribed SFS Turbo resources will not be reclaimed.

If high data security is required, you are advised to select **Retain** to prevent data from being deleted by mistake.

After the definition is complete, run the **kubectl create** commands to create storage resources.

```
# kubectl create -f sas.yaml
storageclass.storage.k8s.io/csi-disk-sas created
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
```

Query the storage class again. Two more types of storage classes are displayed in the command output, as shown below.

```
# kubectl get sc
NAME                PROVISIONER                     AGE
csi-disk            everest-csi-provisioner         17d
csi-disk-sas        everest-csi-provisioner         2m28s
csi-disk-ssd        everest-csi-provisioner         16s
csi-disk-topology   everest-csi-provisioner         17d
csi-nas             everest-csi-provisioner         17d
csi-obs             everest-csi-provisioner         17d
csi-sfsturbo        everest-csi-provisioner         17d
```

Other types of storage resources can be defined in the similar way. You can use kubectl to obtain the YAML file and modify it as required.

- File storage


```
# kubectl get sc csi-nas -oyaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-nas
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: nas.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/share-access-level: rw
  everest.io/share-access-to: 5e3864c6-e78d-4d00-b6fd-de09d432c632   # ID of the VPC to which the cluster belongs
  everest.io/share-is-public: 'false'
  everest.io/zone: xxxxx          # AZ
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

- Object storage

```
# kubectl get sc csi-obs -oyaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-obs
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: s3fs          # Object storage type. s3fs indicates an object bucket, and obsfs indicates a parallel file system.
  everest.io/obs-volume-type: STANDARD      # Storage class of the OBS bucket
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Specifying an Enterprise Project for Storage Classes

CCE allows you to specify an enterprise project when creating EVS disks and OBS PVCs. The created storage resources (EVS disks and OBS) belong to the specified enterprise project. **The enterprise project can be the enterprise project to which the cluster belongs or the default enterprise project.**

If you do not specify any enterprise project, the enterprise project in StorageClass is used by default. The created storage resources by using the csi-disk and csi-obs storage classes of CCE belong to the default enterprise project.

If you want the storage resources created from the storage classes to be in the same enterprise project as the cluster, you can customize a storage class and specify the enterprise project ID, as shown below.

NOTE

To use this function, the everest add-on must be upgraded to 1.2.33 or later.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk-epid      #Customize a storage class name.
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/enterprise-project-id: 86bfc701-9d9e-4871-a318-6385aa368183  #Specify the enterprise project ID.
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```


Specifying a Default Storage Class

You can specify a storage class as the default class. In this way, if you do not specify **storageClassName** when creating a PVC, the PVC is created using the default storage class.

For example, to specify **csi-disk-ssd** as the default storage class, edit your YAML file as follows:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"   # Specifies the default storage class in a cluster. A cluster can have only one default storage class.
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

Delete the created csi-disk-ssd disk, run the **kubectl create** command to create a csi-disk-ssd disk again, and then query the storage class. The following information is displayed.

```
# kubectl delete sc csi-disk-ssd
storageclass.storage.k8s.io "csi-disk-ssd" deleted
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
# kubectl get sc
NAME                      PROVISIONER                     AGE
csi-disk                  everest-csi-provisioner         17d
csi-disk-sas              everest-csi-provisioner         114m
csi-disk-ssd (default)    everest-csi-provisioner         9s
csi-disk-topology         everest-csi-provisioner         17d
csi-nas                   everest-csi-provisioner         17d
csi-obs                   everest-csi-provisioner         17d
csi-sfsturbo              everest-csi-provisioner         17d
```

Verification

- Use **csi-disk-sas** to create a PVC.

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  sas-disk
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
    storageClassName: csi-disk-sas
  ```

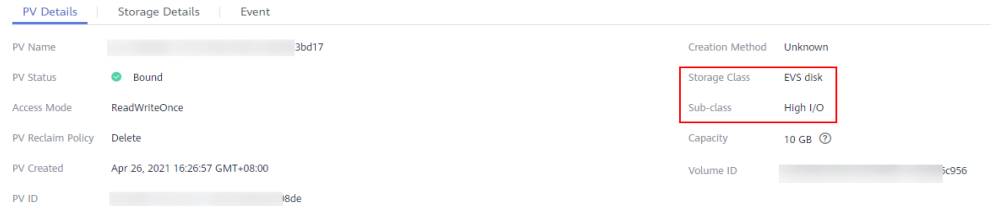
  Create a storage class and view its details. As shown below, the object can be created and the value of **STORAGECLASS** is **csi-disk-sas**.

  ```
  # kubectl create -f sas-disk.yaml
  persistentvolumeclaim/sas-disk created
  # kubectl get pvc
  NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
  STORAGECLASS   AGE
  ```



```
sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            csi-disk-sas
24s
# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                    STORAGECLASS   REASON   AGE
pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            Delete           Bound       default/
sas-disk         csi-disk-sas           30s
```

View the PVC details on the CCE console. On the PV details page, you can see that the disk type is high I/O.



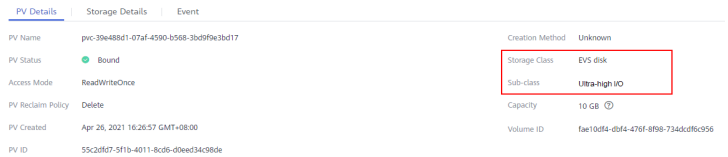
- If **storageClassName** is not specified, the default configuration is used, as shown below.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name:  ssd-disk
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Create and view the storage resource. You can see that the storage class of PVC ssd-disk is csi-disk-ssd, indicating that csi-disk-ssd is used by default.

```
# kubectl create -f ssd-disk.yaml
persistentvolumeclaim/ssd-disk created
# kubectl get pvc
NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            csi-disk-sas
16m
ssd-disk   Bound    pvc-4d2b059c-0d6c-44af-9994-f74d01c78731   10Gi       RWO            csi-disk-ssd
10s
# kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                    STORAGECLASS   REASON   AGE
pvc-4d2b059c-0d6c-44af-9994-f74d01c78731   10Gi       RWO            Delete           Bound
default/ssd-disk         csi-disk-ssd           15s
pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO            Delete           Bound       default/
sas-disk         csi-disk-sas           17m
```

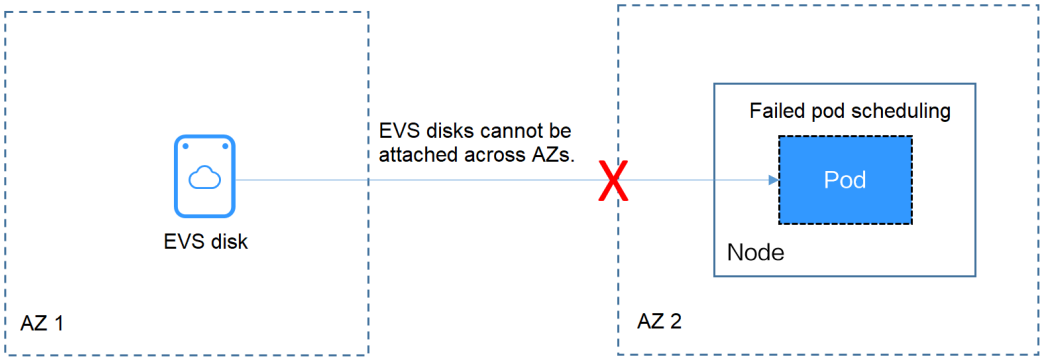
View the PVC details on the CCE console. On the PV details page, you can see that the disk type is ultra-high I/O.



11.6 Enabling Automatic Topology for EVS Disks When Nodes Are Deployed in Different AZs (csi-disk-topology)

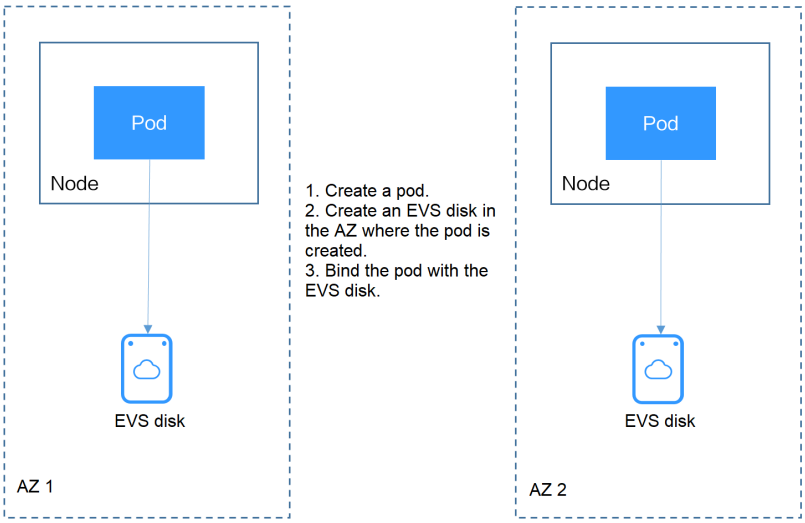
Background

EVS disks cannot be attached to a node deployed in another AZ. For example, the EVS disks in AZ 1 cannot be attached to a node in AZ 2. If the storage class csi-disk is used for StatefulSets, when a StatefulSet is scheduled, a PVC and a PV are created immediately (an EVS disk is created along with the PV), and then the PVC is bound to the PV. However, when the cluster nodes are located in multiple AZs, the EVS disk created by the PVC and the node to which the pods are scheduled may be in different AZs. As a result, the pods fail to be scheduled.



Solution

CCE provides a storage class named **csi-disk-topology**, which is a late-binding EVS disk type. When you use this storage class to create a PVC, no PV will be created in pace with the PVC. Instead, the PV is created in the AZ of the node where the pod will be scheduled. An EVS disk is then created in the same AZ to ensure that the EVS disk can be attached and the pod can be successfully scheduled.



Failed Pod Scheduling Due to csi-disk Used in Cross-AZ Node Deployment

Create a cluster with three nodes in different AZs.

Use the csi-disk storage class to create a StatefulSet and check whether the workload is successfully created.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx                    # Name of the headless Service
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 600m
              memory: 200Mi
            requests:
              cpu: 600m
              memory: 200Mi
          volumeMounts:                 # Storage mounted to the pod
          - name:  data
            mountPath: /usr/share/nginx/html      # Mount the storage to /usr/share/nginx/html.
      imagePullSecrets:
        - name: default-secret
  volumeClaimTemplates:
  - metadata:
      name: data
      annotations:
        everest.io/disk-volume-type: SAS
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: csi-disk
```

The StatefulSet uses the following headless Service.

```
apiVersion: v1
kind: Service       # Object type (Service)
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx     # Name of the port for communication between pods
      port: 80        # Port number for communication between pods
  selector:
    app: nginx        # Select the pod whose label is app:nginx.
  clusterIP: None     # Set this parameter to None, indicating the headless Service.
```

After the creation, check the PVC and pod status. In the following output, the PVC has been created and bound successfully, and a pod is in the Pending state.


```
# kubectl get pvc -owide
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS
AGE   VOLUMEMODE
data-nginx-0   Bound    pvc-04e25985-fc93-4254-92a1-1085ce19d31e   1Gi        RWO            csi-disk
64s   Filesystem
data-nginx-1   Bound    pvc-0ae6336b-a2ea-4ddc-8f63-cfc5f9efe189   1Gi        RWO            csi-disk
47s   Filesystem
data-nginx-2   Bound    pvc-aa46f452-cc5b-4dbd-825a-da68c858720d   1Gi        RWO            csi-disk
30s   Filesystem
data-nginx-3   Bound    pvc-3d60e532-ff31-42df-9e78-015cacb18a0b   1Gi        RWO            csi-disk
14s   Filesystem

# kubectl get pod -owide
NAME      READY   STATUS    RESTARTS   AGE     IP           NODE           NOMINATED NODE   READINESS
GATES
nginx-0   1/1     Running   0          2m25s   172.16.0.12   192.168.0.121   <none>           <none>
nginx-1   1/1     Running   0          2m8s    172.16.0.136   192.168.0.211   <none>           <none>
nginx-2   1/1     Running   0          111s    172.16.1.7     192.168.0.240   <none>           <none>
nginx-3   0/1     Pending   0          95s     <none>         <none>          <none>           <none>
```

The event information of the pod shows that the scheduling fails due to no available node. Two nodes (in AZ 1 and AZ 2) do not have sufficient CPUs, and the created EVS disk is not in the AZ where the third node (in AZ 3) is located. As a result, the pod cannot use the EVS disk.

```
# kubectl describe pod nginx-3
Name:           nginx-3
...
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  111s  default-scheduler  0/3 nodes are available: 3 pod has unbound
immediate PersistentVolumeClaims.
  Warning  FailedScheduling  111s  default-scheduler  0/3 nodes are available: 3 pod has unbound
immediate PersistentVolumeClaims.
  Warning  FailedScheduling  28s   default-scheduler  0/3 nodes are available: 1 node(s) had volume node
affinity conflict, 2 Insufficient cpu.
```

Check the AZ where the EVS disk created from the PVC is located. It is found that data-nginx-3 is in AZ 1. In this case, the node in AZ 1 has no resources, and only the node in AZ 3 has CPU resources. As a result, the scheduling fails. Therefore, there should be a delay between creating the PVC and binding the PV.

Storage Class for Delayed Binding

If you check the cluster storage class, you can see that the binding mode of csi-disk-topology is **WaitForFirstConsumer**, indicating that a PV is created and bound when a pod uses the PVC. That is, the PV and the underlying storage resources are created based on the pod information.

```
# kubectl get storageclass
NAME                PROVISIONER                  RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
csi-disk            everest-csi-provisioner      Delete          Immediate              true                   156m
csi-disk-topology   everest-csi-provisioner      Delete          WaitForFirstConsumer   true
156m
csi-nas             everest-csi-provisioner      Delete          Immediate              true                   156m
csi-obs             everest-csi-provisioner      Delete          Immediate              false                  156m
csi-sfsturbo        everest-csi-provisioner      Delete          Immediate              true                   156m
```

VOLUMEBINDINGMODE is displayed if your cluster is v1.19. It is not displayed in clusters of v1.17 or v1.15.

You can also view the binding mode in the csi-disk-topology details.


```
# kubectl describe sc csi-disk-topology
Name:                  csi-disk-topology
IsDefaultClass:        No
Annotations:           <none>
Provisioner:           everest-csi-provisioner
Parameters:            csi.storage.k8s.io/csi-driver-name=disk.csi.everest.io,csi.storage.k8s.io/
fstype=ext4,everest.io/disk-volume-type=SAS,everest.io/passthrough=true
AllowVolumeExpansion:  True
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

Create PVCs of the csi-disk and csi-disk-topology classes. Observe the differences between these two types of PVCs.

- csi-disk

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  disk
    annotations:
      everest.io/disk-volume-type: SAS
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
    storageClassName: csi-disk        # StorageClass
  ```

- csi-disk-topology

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name:  topology
    annotations:
      everest.io/disk-volume-type: SAS
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 10Gi
    storageClassName: csi-disk-topology        # StorageClass
  ```

View the PVC details. As shown below, the csi-disk PVC is in Bound state and the csi-disk-topology PVC is in Pending state.

```
# kubectl create -f pvc1.yaml
persistentvolumeclaim/disk created
# kubectl create -f pvc2.yaml
persistentvolumeclaim/topology created
# kubectl get pvc
NAME          STATUS    VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS       AGE
disk          Bound     pvc-88d96508-d246-422e-91f0-8caf414001fc   10Gi       RWO            csi-disk
18s
topology      Pending                                                                         csi-disk-topology   2s
```

View details about the csi-disk-topology PVC. You can see that "waiting for first consumer to be created before binding" is displayed in the event, indicating that the PVC is bound after the consumer (pod) is created.

```
# kubectl describe pvc topology
Name:          topology
Namespace:     default
StorageClass:  csi-disk-topology
Status:        Pending
```



```
Volume:
Labels:        <none>
Annotations:   everest.io/disk-volume-type: SAS
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
VolumeMode:    Filesystem
Used By:       <none>
Events:
  Type    Reason                 Age                From                         Message
  ----    ------                 ----               ----                         -------
  Normal  WaitForFirstConsumer  5s (x3 over 30s)  persistentvolume-controller  waiting for first
consumer to be created before binding
```

Create a workload that uses the PVC. Set the PVC name to **topology**.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx:alpine
        name: container-0
        volumeMounts:
        - mountPath: /tmp                                # Mount path
          name: topology-example
      restartPolicy: Always
      volumes:
      - name: topology-example
        persistentVolumeClaim:
          claimName:  topology                        # PVC name
```

After the PVC is created, check the PVC details. You can see that the PVC is bound successfully.

```
# kubectl describe pvc topology
Name:          topology
Namespace:     default
StorageClass:  csi-disk-topology
Status:        Bound
....
Used By:       nginx-deployment-fcd9fd98b-x6tbs
Events:
  Type    Reason                 Age
From                                                                                                  Message
  ----    ------                 ----
----                                                                                                  -------
  Normal  WaitForFirstConsumer   84s (x26 over 7m34s)  persistentvolume-
controller                                                                          waiting for first consumer to be created before
binding
  Normal  Provisioning           54s                   everest-csi-provisioner_everest-csi-
controller-7965dc48c4-5k799_2a6b513e-f01f-4e77-af21-6d7f8d4dbc98  External provisioner is provisioning
volume for claim "default/topology"
  Normal  ProvisioningSucceeded  52s                   everest-csi-provisioner_everest-csi-
controller-7965dc48c4-5k799_2a6b513e-f01f-4e77-af21-6d7f8d4dbc98  Successfully provisioned volume
pvc-9a89ea12-4708-4c71-8ec5-97981da032c9
```

Using csi-disk-topology in Cross-AZ Node Deployment

The following uses csi-disk-topology to create a StatefulSet with the same configurations used in the preceding example.

```
volumeClaimTemplates:
- metadata:
    name: data
    annotations:
      everest.io/disk-volume-type: SAS
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 1Gi
    storageClassName: csi-disk-topology
```

After the creation, check the PVC and pod status. As shown in the following output, the PVC and pod can be created successfully. The nginx-3 pod is created on the node in AZ 3.

```
# kubectl get pvc -owide
NAME           STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS       AGE   VOLUMEMODE
data-nginx-0   Bound    pvc-43802cec-cf78-4876-bcca-e041618f2470   1Gi        RWO           csi-disk-
topology   55s   Filesystem
data-nginx-1   Bound    pvc-fc942a73-45d3-476b-95d4-1eb94bf19f1f   1Gi        RWO           csi-disk-
topology   39s   Filesystem
data-nginx-2   Bound    pvc-d219f4b7-e7cb-4832-a3ae-01ad689e364e   1Gi        RWO           csi-disk-
topology   22s   Filesystem
data-nginx-3   Bound    pvc-b54a61e1-1c0f-42b1-9951-410ebd326a4d   1Gi        RWO           csi-disk-
topology   9s    Filesystem

# kubectl get pod -owide
NAME      READY   STATUS    RESTARTS   AGE   IP            NODE            NOMINATED NODE   READINESS
GATES
nginx-0   1/1     Running   0          65s   172.16.1.8    192.168.0.240   <none>           <none>
nginx-1   1/1     Running   0          49s   172.16.0.13   192.168.0.121   <none>           <none>
nginx-2   1/1     Running   0          32s   172.16.0.137  192.168.0.211   <none>           <none>
nginx-3   1/1     Running   0          19s   172.16.1.9    192.168.0.240   <none>           <none>
```


12 Container

12.1 Properly Allocating Container Computing Resources

If a node has sufficient memory resources, a container on this node can use more memory resources than requested, but no more than limited. If the memory allocated to a container exceeds the upper limit, the container is stopped first. If the container continuously uses memory resources more than limited, the container is terminated. If a stopped container is allowed to be restarted, kubelet will restart it, but other types of run errors will occur.

Scenario 1

The node's memory has reached the memory limit reserved for the node. As a result, OOM killer is triggered.

Solution

You can either scale up the node or migrate the pods on the node to other nodes.

Scenario 2

The upper limit of resources configured for the pod is too small. When the actual usage exceeds the limit, OOM killer is triggered.

Solution

Set a higher upper limit for the workload.

Example

A pod will be created and allocated memory that exceeds the limit. As shown in the following configuration file of the pod, the pod requests 50 MiB memory and the memory limit is set to 100 MiB.

Example YAML file (memory-request-limit-2.yaml):

```
apiVersion: v1
kind: Pod
```



```
metadata:
  name: memory-demo-2
spec:
  containers:
  - name: memory-demo-2-ctr
    image: vish/stress
    resources:
      requests:
        memory: 50Mi
      limits:
        memory: "100Mi"
    args:
    - -mem-total
    - 250Mi
    - -mem-alloc-size
    - 10Mi
    - -mem-alloc-sleep
    - 1s
```

The **args** parameters indicate that the container attempts to request 250 MiB memory, which exceeds the pod's upper limit (100 MiB).

Creating a pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit-2.yaml --
namespace=mem-example
```

Viewing the details about the pod:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

In this stage, the container may be running or be killed. If the container is not killed, repeat the previous command until the container is killed.

```
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          24s
```

Viewing detailed information about the container:

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-example
```

This output indicates that the container is killed because the memory limit is exceeded.

```
lastState:
    terminated:
      containerID: docker://7aae52677a4542917c23b10fb56fcb2434c2e8427bc956065183c1879cc0dbd2
      exitCode: 137
      finishedAt: 2020-02-20T17:35:12Z
      reason: OOMKilled
      startedAt: null
```

In this example, the container can be automatically restarted. Therefore, kubelet will start it again. You can run the following command several times to see how the container is killed and started:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

The preceding command output indicates how the container is killed and started back and forth:

```
$ kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS      RESTARTS   AGE
memory-demo-2   0/1       OOMKilled   1          37s
$ kubectl get pod memory-demo-2 --namespace=mem-example
NAME            READY     STATUS    RESTARTS   AGE
memory-demo-2   1/1       Running   2          40s
```

Viewing the historical information of the pod:


```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

The following command output indicates that the pod is repeatedly killed and started.

```
... Normal  Created   Created container with id
66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511
... Warning BackOff   Back-off restarting failed container
```

12.2 Upgrading Pods Without Interrupting Services

Application Scenarios

In a Kubernetes cluster, applications can be accessed externally through Deployments and LoadBalancer Services. When an application is updated or upgraded, new pods are created in the Deployment. These new pods will gradually replace the old ones. During this process, services may be interrupted.

Solution

To prevent an application upgrade from interrupting services, configure Deployments and Services as follows:

- In a Deployment, upgrade pods in the **Rolling upgrade** mode. In this mode, pods are updated one by one, not all at once. In this way, you can control the update speed and the number of concurrent pods to ensure that services are not interrupted during the upgrade. For example, you can configure the **maxSurge** and **maxUnavailable** parameters to control the number of new pods created and the number of old pods deleted concurrently. Ensure that there is always a workload that can provide services during the upgrade.
- There are two types of service affinity in a LoadBalancer:
  - **Cluster-level** service affinity (**externalTrafficPolicy: Cluster**). In this mode, if there is no pod deployed on a node, the request is forwarded to pods on another node. During the cross-node forwarding, the source IP address may be lost.
  - **Node-level** service affinity (**externalTrafficPolicy: Local**). In this mode, requests are directly forwarded to the node where the pod resides. Cross-node forwarding is not involved. Therefore, the source IP address can be preserved. However, if the node where the pod resides changes during the rolling upgrade, the ELB backend server will change accordingly, which may cause service interruption. In this case, you can upgrade pods in place. This ensures that there is at least one pod running properly on the ELB backend node.

The following table lists the solution for ensuring service continuity during a pod upgrade.

Scenario	Service	Deployment
The source IP address does not need to be preserved.	Select the **Cluster-level** service affinity.	Select **Rolling upgrade** for **Upgrade Mode**, configure a graceful termination, and enable **Liveness probe** and **Ready probe**.
The source IP address needs to be preserved.	Select the **Node-level** service affinity.	Select **Rolling upgrade** for **Upgrade Mode**, configure a graceful termination, enable **Liveness probe** and **Ready probe**, and add **Node Affinity** policies. (Ensure that there is at least one pod running on each node during the update.)

Procedure

In this example, there are 200 replicas in the workload, and the workload is exposed through the LoadBalance Service. The rolling upgrade of workloads associated with Loadbalance or Ingress Services involves multiple Services. Therefore, you need to pay attention to the configuration of rolling upgrade parameters.

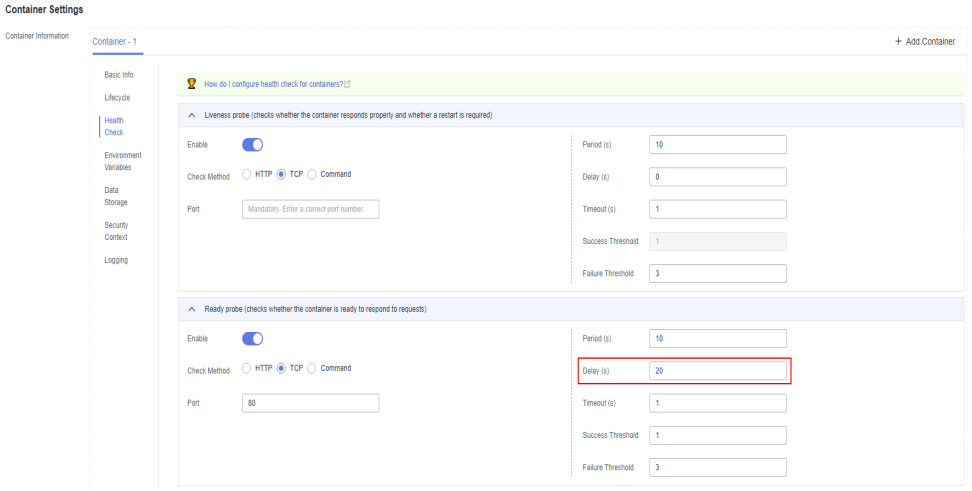
Step 1 Log in to the CCE console and click the cluster name to access the cluster console. In the navigation pane, choose **Workloads**.

Step 2 In the workload list, click **Upgrade** in the **Operation** column of the workload to be upgraded. The **Upgrade Workload** page is displayed.

1. Enable the liveness probe and ready probe. In the **Container Settings** area, click **Health Check** and enable **Liveness probe** and **Ready probe**. In this example, **TCP** is selected for **Check Method**. Configure the parameters based on your requirements. Parameters like **Period (s)**, **Delay (s)**, and **Timeout (s)** must be properly configured. Some applications take a long time to start. A small value of these parameters will lead to repeated restart.

   In this example, the ready probe delay is set to **20** to control the interval for rolling workloads in batches.

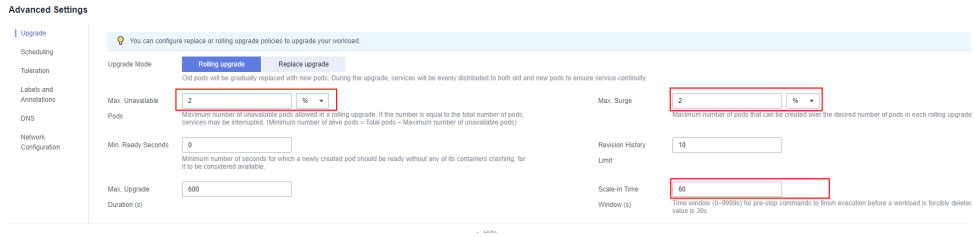
Figure 12-1 Enabling the liveness probe and ready probe



2. Configure a rolling upgrade. In the **Advanced Settings** area, click **Upgrade** and select **Rolling upgrade** for **Upgrade Mode**. This ensures that the instances of the old versions are gradually replaced with the ones of the new versions.

   In this example, **maxUnavailable** is set to **2%**, and **maxSurge** is set to **2%** to control the workload rolling step. This, works with the ready probe delay, enables eight workloads to be upgraded every 20 seconds.

Figure 12-2 Configuring a rolling upgrade

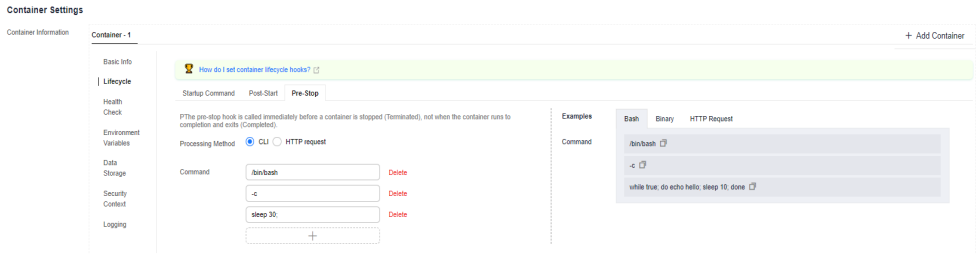


3. Configure a graceful termination.

   a. In the **Container Settings** area, click **Lifecycle** and configure pre-stop processing. Configure this parameter to the time required for the Service to process all remaining requests, most of which are persistent connection requests. You can, for example, set the workload to hibernate for 30s after receiving a deletion request so that the workload can have sufficient time to process the remaining requests to ensure proper service running.

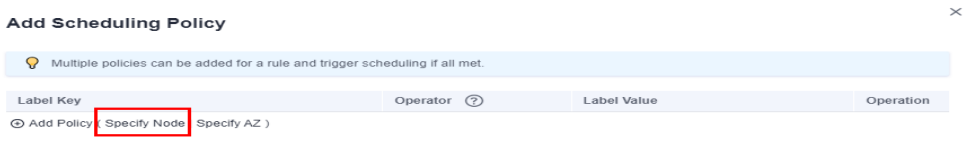
   b. In the **Advanced Settings** area, click **Upgrade**. Configure **Scale-In Time Window (terminationGracePeriodSeconds)** to specify the waiting time for command execution before the container is stopped. The scale-in time window must be greater than the pre-stop processing time. Add 30s to the command execution time before the container is stopped. If, for example, the pre-stop processing time is 30s, the scale-in time window should be 60s.

Figure 12-3 Entering the pre-stop command



4. Add node affinity policies. Add this kind of policy when **Node-level** is selected for a Service's **Service Affinity**. In the **Advanced Settings** area, click **Scheduling** and add **Node Affinity** policies. When adding a scheduling policy, specify the nodes that the workload requires affinity.

Figure 12-4 Adding node affinity policies



Step 3 After the configuration is complete, click **Upgrade Workload**.

On the **Pods** tab, after a newly created pod is displayed, stop the old one. This ensures that there is always a pod running in the workload.

----End

12.3 Modifying Kernel Parameters Using a Privileged Container

Prerequisites

To access a Kubernetes cluster from a client, you can use the Kubernetes command line tool kubectl. For details, see **Connecting to a Cluster Using kubectl**.

Procedure

Step 1 Create a DaemonSet in the background, select the Nginx image, enable the Privileged Container, configure the lifecycle, and add the **hostNetwork** field (value: **true**).

1. Create a **daemonSet** file.

   vi daemonSet.yaml

   An example YAML file is provided as follows:

NOTICE

The **spec.spec.containers.lifecycle** field indicates the command that will be run after the container is started.

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: daemonset-test
  labels:
    name: daemonset-test
spec:
  selector:
    matchLabels:
      name: daemonset-test
  template:
    metadata:
      labels:
        name: daemonset-test
    spec:
      hostNetwork: true
      containers:
      - name: daemonset-test
        image: nginx:alpine-perl
        command:
        - "/bin/sh"
        args:
        - "-c"
        - while :; do  time=$(date);done
        imagePullPolicy: IfNotPresent
        lifecycle:
          postStart:
            exec:
              command:
              - sysctl
              - "-w"
              - net.ipv4.tcp_tw_reuse=1
        securityContext:
          privileged: true
      imagePullSecrets:
      - name: default-secret
```

2. Create a DaemonSet.

   kubectl create –f daemonSet.yaml

Step 2 Check whether the DaemonSet is successfully created.

kubectl get daemonset *DaemonSet name*

In this example, run the following command:

kubectl get daemonset daemonset-test

Information similar to the following is displayed:

```
NAME             DESIRED   CURRENT   READY   UP-T0-DATE   AVAILABLE   NODE SELECTOR   AGE
daemonset-test   2         2         2       2            2           <node>          2h
```

Step 3 Query the container ID of DaemonSet on the node.

docker ps -a|grep *DaemonSet name*

In this example, run the following command:

docker ps -a|grep daemonset-test

Information similar to the following is displayed:

```
897b99faa9ce        3e094d5696c1                        "/bin/sh  -c while..."     31 minutes ago      Up  30
minutes  ault_fa7cc313-4ac1-11e9-a716-fa163e0aalba_0
```

Step 4 Access the container.

docker exec -it *containerid* **/bin/sh**

In this example, run the following command:

docker exec -it *897b99faa9ce* **/bin/sh**

Step 5 Check whether the configured command is executed after the container is started.

sysctl -a |grep net.ipv4.tcp_tw_reuse

If the following information is displayed, the system parameters are modified successfully:

```
net.ipv4.tcp_tw_reuse=1
```

----End

12.4 Using Init Containers to Initialize an Application

Concepts

Before containers running applications are started, one or some init containers are started first. If there are multiple init containers, they will be started in the defined sequence. The application containers are started only after all init containers run to completion and exit. Storage volumes in a pod are shared. Therefore, the data generated in the init containers can be used by the application containers.

Init containers can be used in multiple Kubernetes resources, such as Deployments, DaemonSets, and jobs. They perform initialization before application containers are started.

Application Scenarios

Before deploying a service, you can use an init container to make preparations before the pod where the service is running is deployed. After the preparations are complete, the init container runs to completion and exit, and the container to be deployed will be started.

- **Scenario 1: Wait for other modules to be ready.** For example, an application contains two containerized services: web server and database. The web server service needs to access the database service. However, when the application is started, the database service may have not been started. Therefore, web server may fail to access database. To solve this problem, you can use an init container in the pod where web server is running to check whether database is ready. The init container runs to completion only when database is accessible. Then, web server is started and initiates a formal access request to database.
- **Scenario 2: Initialize the configuration.** For example, the init container can check all existing member nodes in the cluster and prepare the cluster configuration information for the application container. After the application

container is started, it can be added to the cluster using the configuration information.

- **Other scenarios**: For example, register a pod with a central database and download application dependencies.

For details, see **Init Containers**.

Procedure

Step 1 Edit the YAML file of the init container workload.

vi deployment.yaml

An example YAML file is provided as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      name: mysql
  template:
    metadata:
      labels:
        name: mysql
    spec:
      initContainers:
      - name: getresource
        image: busybox
        command: ['sleep 20']
      containers:
      - name: mysql
        image: percona:5.7.22
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
        resources:
          limits:
            memory: "500Mi"
            cpu: "500m"
          requests:
            memory: "500Mi"
            cpu: "250m"
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "mysql"
```

Step 2 Create an init container workload.

kubectl create -f deployment.yaml

Information similar to the following is displayed:

```
deployment.apps/mysql created
```

Step 3 Query the created Docker container on the node where the workload is running.

docker ps -a|grep mysql

The init container will exit after it runs to completion. The query result **Exited (0)** shows the exit status of the init container.


```
0dc822969e3f        percona             "docker-entrypoint..."   34 seconds ago       Up 33 seconds
ql_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
a745881214e7        busybox             "sh -c 'sleep 20'"       About a minute ago   Exited (0) 50 seconds ago
resource_mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
615db9e60a80        cfe-pause:1l.23.1   "/pause"                 About a minute ago   Up About a minute
 mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
```

----End

12.5 Setting Time Zone Synchronization

Case Scenarios

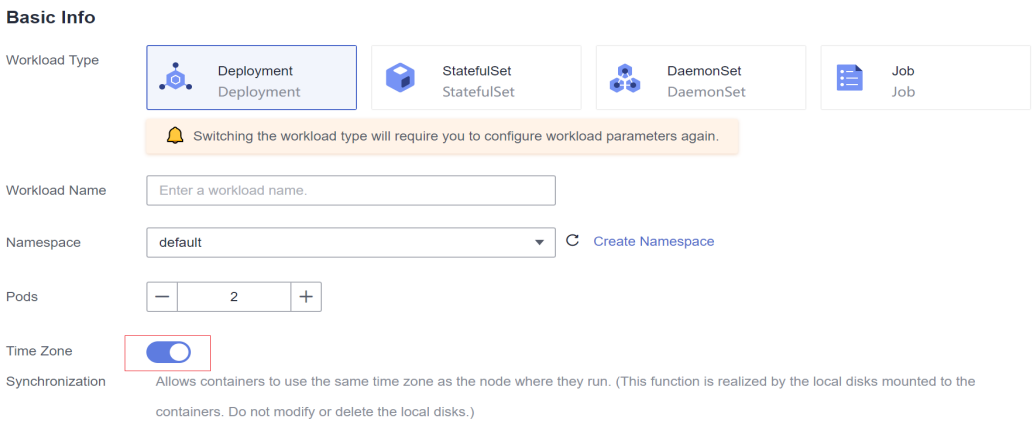
- Scenario 1: Setting Time Zone Synchronization Between Containers and Nodes
- Scenario 2: Setting Time Zone Synchronization Among Containers, Container Logs, and Nodes
- Scenario 3: Setting Time Zone Synchronization Between Workloads and Nodes

Scenario 1: Setting Time Zone Synchronization Between Containers and Nodes

Step 1 Log in to the CCE console.

Step 2 In the **Basic Info** area of the **Create Workload** page, enable **Time Zone Synchronization** so that the same time zone will be used for both the container and the node.

Figure 12-5 Enabling the time zone synchronization



Step 3 Log in to the node, go to the container, and check whether the time zone of the container is the same as that of the node.

date -R

Information similar to the following is displayed:

```
Tue, 04 Jun 2019 15::08:47  +0800
```

docker ps -a|grep *test*

Information similar to the following is displayed:


```
0edd74c66bdb        b2b9b536b744        "nginx -g 'daemon ..."   6 hours ago         Up 6 hours
                                k8s_container-0_test-7d7d7f4965-xwqkx_default_abf6df2e-85f7-11e9-93df-fa163ee0f9
1a_1
```

docker exec -it *oedd74c66bdb* **/bin/sh**

date -R

Information similar to the following is displayed:

```
Tue, 04 Jun 2019 15:09:20  +0800
```

----End

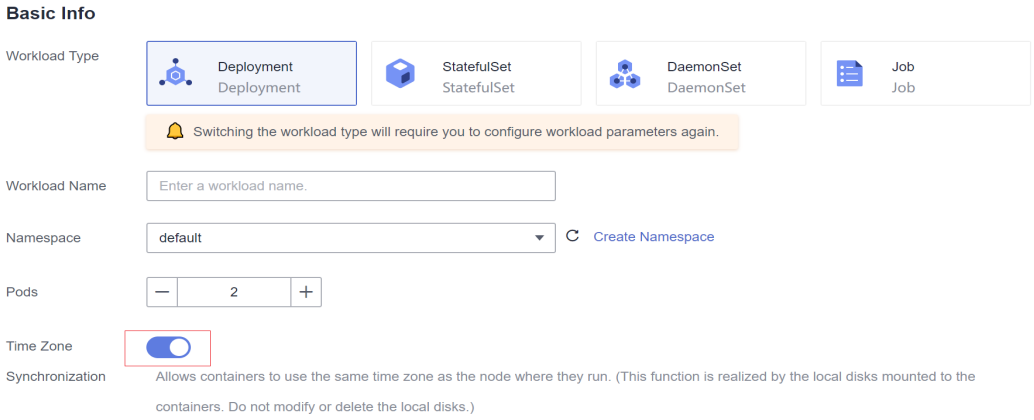
Scenario 2: Setting Time Zone Synchronization Among Containers, Container Logs, and Nodes

The difference between the time when the Java application prints logs and the container's standard time obtained in date -R mode is 8 hours.

Step 1 Log in to the CCE console.

Step 2 In the **Basic Info** area of the **Create Workload** page, enable **Time Zone Synchronization** so that the same time zone will be used for both the container and the node.

Figure 12-6 Enabling the time zone synchronization



Step 3 Log in to the node, go to the container, and modify the **catalina.sh** script.

cd /usr/local/tomcat/bin

vi catalina.sh

If you cannot run the **vi** command in the container, go to **Step 4** or run the **vi** command to add **-Duser.timezone=GMT+08** to the script, as shown in the following figure.

```
# Do this here so custom URL handles (specifically 'war:...') can be used in the security policy
JAVA_OPTS="$JAVA_OPTS -Djava.protocol.handler.pkgs=org.apache.catalina.webresources -Duser.timezone=GMT+08"
```

Step 4 Copy the script from the container to the node, add **-Duser.timezone=GMT+08** to the script, and then copy the script from the node to the container.

Run the following command to copy files in the container to the host machine:

docker cp mycontainer: /usr/local/tomcat/bin/catalina.sh /home/catalina.sh

Run the following command to copy files from the host machine to the container:

docker cp /home/catalina.sh mycontainer:/ usr/local/tomcat/bin/catalina.sh

Step 5 Restart the container.

docker restart *container_id*

Step 6 Check whether the time zone of the logs is the same as that of the node.

On the CCE console, click the workload name. On the workload details page displayed, click **Logs** in the upper right corner to view the log details. It takes about 5 minutes to load the logs.

----End

Scenario 3: Setting Time Zone Synchronization Between Workloads and Nodes

- Method 1: Set the time zone to CST when creating a container image.
- Method 2: If you do not want to modify the container, when creating a workload on the CCE console, mount the **/etc/localtime** directory of the local host to the **/etc/localtime** directory of the container.

  Example:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: test
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      volumes:
        - name: vol-162979628557461404
          hostPath:
            path: /etc/localtime
            type: ''
      containers:
        - name: container-0
          image: 'nginx:alpine'
          volumeMounts:
            - name: vol-162979628557461404
              readOnly: true
              mountPath: /etc/localtime
          imagePullPolicy: IfNotPresent
      imagePullSecrets:
        - name: default-secret
```


12.6 Setting the Container Network Bandwidth Limit

Application Scenarios

Containers on the same node share the host network bandwidth. Limiting the network bandwidth of containers can effectively prevent mutual interference between containers and improve container network stability.

Constraints

The following shows constraints on setting the rate limiting for inter-pod access:

Constraint Type	Tunnel network model	VPC network model	Cloud Native 2.0 Network Model
Supported versions	All versions	Clusters of v1.19.10 and later	Clusters of v1.19.10 and later
Supported runtime types	Only common containers		
Supported pod types	Only non-HostNetwork pods		
Supported scenarios	Inter-pod access, pods accessing nodes, and pods accessing services		
Constraints	None	None	• Pods access external cloud service CIDR blocks 100.64.0.0/10 and 214.0.0.0/8. • Traffic rate limiting of health check
Value range of rate limit	Only the rate limit in the unit of Mbit/s or Gbit/s is supported, for example, 100 Mbit/s and 1 Gbit/s. The minimum value is 1 Mbit/s and the maximum value is 4.29 Gbit/s.		

Procedure

Step 1 Edit a YAML file for a workload.

vi deployment.yaml

Set the network bandwidth for the pod in spec.template.metadata.**annotations** to limit the network traffic of the container. For details about the network bandwidth limit fields, see **Table 12-1**.

If the parameters are not specified, the network bandwidth is not limited by default.

An example is as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
      annotations:
       # Ingress bandwidth
        kubernetes.io/ingress-bandwidth: 100M
       # Egress bandwidth
        kubernetes.io/egress-bandwidth: 1G
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: nginx
      imagePullSecrets:
      - name: default-secret
```

Table 12-1 Fields for limiting the network bandwidth of pods

Field	Description	Mandatory
kubernetes.io/ingress-bandwidth	Ingress bandwidth for a pod. Value range: 1k-1P. If this field is set to a value greater than 32 Gbit/s, the actual ingress bandwidth that a pod can use is 32 Gbit/s.	No
kubernetes.io/egress-bandwidth	Egress bandwidth for a pod. Value range: 1k-1P. If this field is set to a value greater than 32 Gbit/s, the actual egress bandwidth that a pod can use is 32 Gbit/s.	No

Step 2 Create a workload.

kubectl create -f deployment.yaml

Information similar to the following is displayed:

```
deployment.apps/nginx created
```

----End

12.7 Using hostAliases to Configure /etc/hosts in a Pod

Application Scenarios

If DNS or other related settings are inappropriate, you can use **hostAliases** to overwrite the resolution of the hostname at the pod level when adding entries to the **/etc/hosts** file of the pod.

Procedure

Step 1 Use kubectl to connect to the cluster.

Step 2 Create the **hostaliases-pod.yaml** file.

vi hostaliases-pod.yaml

The field in bold in the YAML file indicates the image name and tag. You can replace the example value as required.

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  hostAliases:
  - ip: 127.0.0.1
    hostnames:
    - foo.local
    - bar.local
  - ip: 10.1.2.3
    hostnames:
    - foo.remote
    - bar.remote
  containers:
    - name: cat-hosts
      image: tomcat:9-jre11-slim
      lifecycle:
        postStart:
          exec:
            command:
              - cat
              - /etc/hosts
  imagePullSecrets:
    - name: default-secret
```

Table 12-2 pod field description

Parameter	Mandatory	Description
apiVersion	Yes	API version number
kind	Yes	Type of the object to be created
metadata	Yes	Metadata definition of a resource object
name	Yes	Name of a pod

Parameter	Mandatory	Description
spec	Yes	Detailed description of the pod. For details, see **Table 12-3**.

Table 12-3 spec field description

Parameter	Mandatory	Description
hostAliases	Yes	Host alias
containers	Yes	For details, see **Table 12-4**.

Table 12-4 containers field description

Parameter	Mandatory	Description
name	Yes	Container name
image	Yes	Container image name
lifecycle	No	Lifecycle

Step 3 Create a pod.

kubectl create -f hostaliases-pod.yaml

If information similar to the following is displayed, the pod is created.

```
pod/hostaliases-pod created
```

Step 4 Query the pod status.

kubectl get pod hostaliases-pod

If the pod is in the **Running** state, the pod is successfully created.

```
NAME              READY     STATUS    RESTARTS   AGE
hostaliases-pod   1/1       Running   0          16m
```

Step 5 Check whether the **hostAliases** functions properly.

docker ps |grep hostaliases-pod

docker exec -ti *Container ID* /bin/sh


```
root@hostaliases-pod:/# cat /etc/hosts
# Kubernetes-managed hosts file.
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.0.0.25       hostaliases-pod

# Entries added by HostAliases.
127.0.0.1       foo.local       bar.local
10.1.2.3        foo.remote      bar.remote
```

----End

12.8 Configuring Domain Name Resolution for CCE Containers

This section describes how to configure domain name resolution for CCE containers.

Service

- Create a Service before you create a workload (Deployment or ReplicaSet). When the Kubernetes starts a container, it provides environment variables that point to all the Services that are running when the container is started. For example, if a Service named foo exists, all containers will obtain the following variables when they are initialized.

  ```
  FOO_SERVICE_HOST=<the host the Service is running on>
  FOO_SERVICE_PORT=<the port the Service is running on>
  ```

  Therefore, you must create a Service first. Otherwise, the environment variables do not take effect. This restriction does not apply to DNS.
- CCE clusters provide the coredns add-on as the DNS server. The DNS server monitors the Kubernetes APIs for the new Services and creates a set of DNS records for each Service. If DNS is enabled throughout the cluster, all pods will be able to automatically resolve the names of Services.
- Do not specify a hostPort for a pod unless necessary. When a pod is bound to a hostPort, the number of locations to which the pod can be scheduled will be limited because each <*hostIP*, *hostPort*, *protocol*> must be unique. If you do not specify **hostIP** and **protocol**, Kubernetes uses 0.0.0.0 as the default host IP address and TCP as the default protocol.

If you only need to access the port for debugging, you can use apiserver proxies or kubectl port-forward.

If you want to open the pod port on the node, consider using the NodePort Service before using hostPort.

- Do not use hostNetwork. The reason is the same as that of using hostPort.
- When kube-proxy load balancing is not required, use headless Services (**ClusterIP** set to **None**) for service discovery.

DNS

By default, CCE provides a DNS add-on Service named coredns to automatically assign DNS domain names for other Services. If it is running in the cluster, run the following command to check the status:

```
kubectl get services coredns --namespace=kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   ClusterIP   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

If the pod is not running, you can run the **describe** command to check why the pod is not started. Assume that there is a Service that has a permanent IP address and a DNS server (coredns cluster add-on) that assigns domain name to the IP address. In this way, any pod in the cluster can communicate with the Service. You can run another application for testing. Enable a new pod, access the pod, and run the **curl** command to check whether the domain name of the Service can be correctly resolved. In some cases, the **curl** command cannot be executed due to the DNS search principles and configuration.

When a pod is created on the CCE console, not all dnsConfig configurations are opened and some default values of the pod domain name resolution parameters are used. You need to know well the default configurations. A typical case is **ndots**. If the number of dots is within the **ndots** threshold range, the domain name is considered as an internal domain name of the Kubernetes cluster and the **..svc.cluster.local** suffix is added to the domain name.

DNS Search Principles and Rules

DNS configuration file: **/etc/resolv.conf**

```
nameserver 10.247.x.x
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:3
```

Parameters:

- **nameserver**: domain name resolution server
- **search**: domain name suffix search rule. More search configurations indicate more matching times for domain name resolution. For example, if three suffixes are matched, at least six search operations are required because both IPv4 and IPv6 addresses need to be checked.
- **options**: domain name resolution option. Multiple KV values are available. A typical case is **ndots**. If the number of dots in the domain name to be accessed exceeds the value of **ndots**, the domain name is considered as a complete domain name and is directly parsed. If the number of dots is less than the value of **ndots**, the suffix **..svc.cluster.local** will be added.

Parameters in Kubernetes dnsConfig

- **nameservers**: a list of IP addresses that will be used as DNS servers for the pod. A maximum of three IP addresses can be specified. If pod's **dnsPolicy** is set to **None**, the list must contain at least one IP address, otherwise this

property is optional. The servers listed will be combined to the base nameservers generated from the specified DNS policy with duplicate addresses removed.

- **searches**: a list of DNS search domains for hostname lookup in the pod. This property is optional. When specified, the provided list will be merged into the base search domain names generated from the chosen DNS policy. Duplicate domain names are removed. Kubernetes allows for at most 6 search domains.
- **options**: an optional list of objects where each object may have a **name** property (required) and a **value** property (optional). The contents in this property will be merged to the options generated from the specified DNS policy. Duplicate entries are removed.

For details, see **DNS for Services and Pods**.

Pod DNS Policies

DNS policies can be set on a per-pod basis. Currently, three types of DNS policies are supported: **Default**, **ClusterFirst**, and **None**.

- **Default**: The DNS configuration of the pod inherits from the host. That is, the DNS configuration of the pod is the same as that of the host and node.
- **ClusterFirst**: Unlike the **Default** policy, the **ClusterFirst** policy writes kube-dns (or CoreDNS) information to the DNS configuration of the pod in advance. **ClusterFirst** is the default pod policy. If **PodPolicy** is not specified for the pod, **dnsPolicy** is preset to **ClusterFirst**. However, **ClusterFirst** is mutually exclusive with **HostNetwork=true**. If **HostNetwork** is set to **true**, the **ClusterFirst** policy will be forcibly changed to the **Default** policy.
- **None**: This policy will clear the DNS configuration preset for the pod. If **dnsPolicy** is set to **None**, Kubernetes does not load any DNS configuration that is determined by its own logic in advance for the pod. Therefore, if you want to set **dnsPolicy** to **None**, you are advised to set **dnsConfig** to describe custom DNS parameters. This setting ensures that the pod has DNS configuration.

DNS configuration scenarios are provided as follows:

Scenario 1: Using a custom DNS

The following example allows you to use a custom DNS to resolve the application domain name configuration in pods. After application migration, you do not need to modify the configuration.

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
```



```
  options:
    - name: ndots
      value: "2"
    - name: edns0
```

Scenario 2: Using the Kubernetes DNS add-on CoreDNS

The DNS service of Kubernetes is preferentially used for domain name resolution. If the resolution fails, the DNS service of an external cascading system is used for domain name resolution.

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: ClusterFirst
```

Scenario 3: Using the public network domain name resolution

This mode applies to the scenario where the domain names in pods are to be accessed from public networks. In this case, the applications in the pods resolve domain names from an external DNS.

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: Default
```

Scenario 4: Using hostNetwork

If **hostNetwork: true** is used to configure the networking in the pod, the network ports of the host machine are exposed to the application running in the pod. All network ports on the LAN where the host machine is located can be used to access the application.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      hostNetwork: true
      dnsPolicy: ClusterFirstWithHostNet
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

If **dnsPolicy: ClusterFirstWithHostNet** is not added, even if the pod uses the DNS of the host machine by default, other pods in the Kubernetes cluster cannot be accessed through the Service name in the container.

CoreDNS Configuration

1. Configuring the CoreDNS ConfigMap

The default CoreDNS configuration file is as follows:

```
Corefile: |
  .:53 {
      errors
      health
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
      }
      prometheus :9153
      forward . /etc/resolv.conf
      cache 30
      loop
      reload
      loadbalance
```

Parameters:

- **error**: Errors are recorded in stdout.
- **health**: The CoreDNS running status report can be obtained from **http://localhost:8080/health**.
- **kubernetes**: The CoreDNS returns a DNS query response based on the IP addresses of the Kubernetes Service and pod.
- **prometheus**: The measurement standard of CoreDNS can be found in the metrics in the format of http://localhost:9153/Prometheus. You can obtain monitoring data in Prometheus format from **http://localhost:9153/metrics**.
- **proxy** and **forward**: Any query that is not in the Kubernetes cluster domain is forwarded to the predefined resolver (/etc/resolv.conf). If the domain name cannot be resolved locally, query the upper-level address. By default, the **/etc/resolv.conf** configuration of the host machine is used.
- **cache**: The front-end cache is enabled.
- **loop**: Simple forwarding loops are detected. If a loop is detected, the CoreDNS process is stopped.
- **reload**: The changed Corefile can be automatically reloaded. After editing the ConfigMap, wait for two minutes for the modification to take effect.
- **loadbalance**: This is a round-robin DNS load balancer that randomizes the order of A, AAAA, and MX records in the answer.

2. Configuring an external DNS server

Some services are not in the Kubernetes environment and need to be accessed through the DNS. The suffix of the service name is **carey.com**.

```
carey.com:53 {
      errors
      cache 30
      proxy . 10.150.0.1
   }
```

Complete configuration file:

```
Corefile: |
  .:53 {
      errors
```



```
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
      pods insecure
      upstream
      fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    forward . /etc/resolv.conf
    cache 30
    loop
    reload
    loadbalance
}
carey.com:53 {
    errors
    cache 30
    proxy . 10.150.0.1
}
```

Currently, CCE add-on management supports the configuration of stub-domains, which is more flexible and convenient than the direct editing of ConfigMaps. You do not need to configure the domain name resolution for pods.

12.9 Using Dual-Architecture Images (x86 and Arm) in CCE

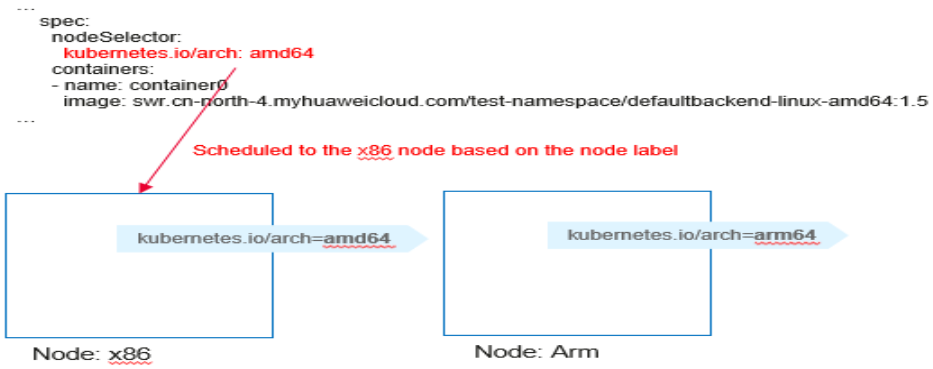
Background

CCE allows you to create x86 and Arm nodes in the same cluster. Due to different underlying architectures, Arm images (applications) cannot run on x86 nodes, and vice versa. As a result, workloads may fail to be deployed in the clusters containing x86 and Arm nodes.

Solution

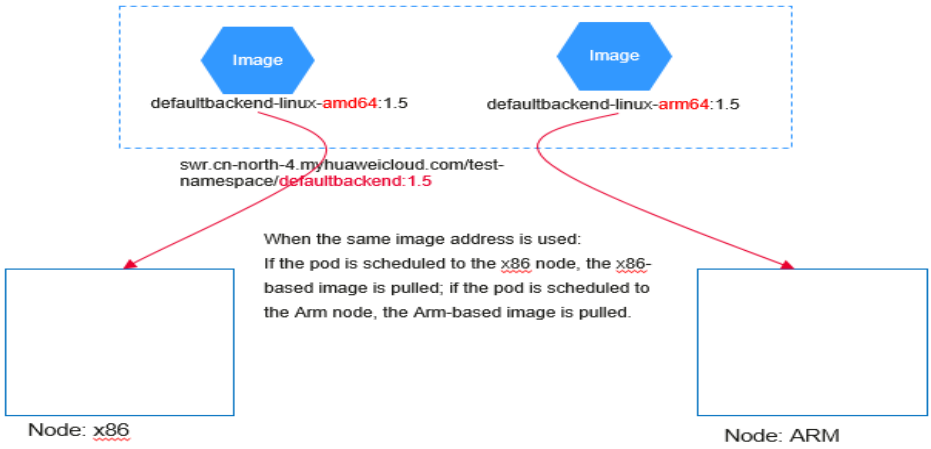
To address this issue, use either of the following methods:

- Set the service affinity when you create a workload so that the pod can be scheduled to an Arm node when the Arm-based image is used or to an x86 node when the x86-based image is used.



- Build a dual-architecture image that supports both x86 and Arm architectures. When a pod is scheduled to an Arm node, the Arm variant in the image is pulled. When a pod is scheduled to an x86 node, the x86 variant in the image is pulled. A dual-architecture image has two variants but has one unified

access path. When deploying a workload, you only need to specify one image path without configuring the service affinity. In this case, the workload description file is simpler and easier to maintain.



Affinity Configuration Description

When creating a node, CCE automatically adds the **kubernetes.io/arch** label to the node to indicate the node architecture.

```
kubernetes.io/arch=amd64
```

The value **amd64** indicates the x86 architecture, and **arm64** indicates the Arm architecture.

When creating a workload, you can configure the node affinity to schedule pods to nodes using the corresponding architecture.

You can use **nodeSelector** in the YAML file to configure the architecture.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test
spec:
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      nodeSelector:
        kubernetes.io/arch: amd64
      containers:
      - name: container0
        image: swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
        resources:
          limits:
            cpu: 250m
            memory: 512Mi
          requests:
            cpu: 250m
            memory: 512Mi
      imagePullSecrets:
      - name: default-secret
```


Building a Dual-Architecture Image

> **NOTE**
>
> To create a dual-architecture image, ensure that the Docker client version is later than 18.03.

The essence of building a dual-architecture image is to build images based on the x86 and Arm architectures separately and then build the dual-architecture image manifest.

For example, the **defaultbackend-linux-amd64:1.5** and **defaultbackend-linux-arm64:1.5** are images based on the x86 and Arm architectures, respectively.

Upload the two images to SWR. For details about how to upload an image, see **Uploading an Image Through a Container Engine Client**.

```
# Add a tag to the original amd64 image defaultbackend-linux-amd64:1.5.
docker tag defaultbackend-linux-amd64:1.5  swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
# Add a tag to the original arm64 image defaultbackend-linux-arm64:1.5.
docker tag defaultbackend-linux-arm64:1.5  swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5
# Push the amd64 image to the image repository.
docker push swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
# Push the arm64 image to the image repository.
docker push swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5
```

Create a dual-architecture **manifest** file and upload it.

```
# Enable DOCKER_CLI_EXPERIMENTAL.
export DOCKER_CLI_EXPERIMENTAL=enabled
# Create the manifest image file.
docker manifest create --amend --insecure swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5 swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5 swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5
# Add arch information to the manifest image file.
docker manifest annotate swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5 swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5 --arch amd64
docker manifest annotate swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5 swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5 --arch arm64
# Push the manifest image file to the image repository.
docker manifest push -p --insecure swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5
```

In this way, you only need to use the image path **swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend:1.5** when creating a workload.

- When a pod is scheduled to an x86 node, the **swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-amd64:1.5** image is pulled.
- When a pod is scheduled to an Arm node, the **swr.ap-southeast-1.myhuaweicloud.com/test-namespace/defaultbackend-linux-arm64:1.5** image is pulled.

12.10 Configuring Core Dumps

Application Scenarios

Linux allows you to create a core dump file if an application crashes, which contains the data the application had in memory at the time of the crash. You can analyze the file to locate the fault.

Generally, when a service application crashes, its container exits and is reclaimed and destroyed. Therefore, container core files need to be permanently stored on the host or cloud storage. This topic describes how to configure container core dumps.

Constraints

When a container core dump is persistently stored to OBS (parallel file system or object bucket), the default mount option **umask=0** is used. As a result, although the core dump file is generated, the core dump information cannot be written to the core file. You can configure the OBS mount option **umask=0077** to store core dump files to OBS. For details, see **Configuring OBS Mount Options**.

Enabling Core Dump on a Node

Log in to the node, run the following command to enable core dump, and set the path and format for storing core files:

echo "/tmp/cores/core.%h.%e.%p.%t" > /proc/sys/kernel/core_pattern

%h, **%e**, **%p**, and **%t** are placeholders, which are described as follows:

- **%h**: hostname (or pod name). You are advised to configure this parameter.
- **%e**: program file name. You are advised to configure this parameter.
- **%p**: (optional) process ID.
- **%t**: (optional) time of the core dump.

After the core dump function is enabled by running the preceding command, the generated core file is named in the format of **core.{*Host name*}.{*Program file name*}.{*Process ID*}.{*Time*}**.

You can also configure a pre-installation or post-installation script to automatically run this command when creating a node.

NOTE

EulerOS 2.3 Systemd has a **bug** that affects container core dump. To use core dump, perform the following operations:

1. In the **/usr/lib/systemd/system/docker.service** file on the node, change the value of **LimitCORE** to **infinity**.
2. Restart Docker.
3. Redeploy service containers.

Permanently Storing Core Dumps

A core file can be stored in your host (using a hostPath volume) or cloud storage (using a PVC). The following is an example YAML file for using a hostPath volume.

```
apiVersion: v1
kind: Pod
metadata:
  name: coredump
spec:
  volumes:
  - name: coredump-path
    hostPath:
      path: /home/coredump
  containers:
  - name: ubuntu
    image: ubuntu:12.04
    command: ["/bin/sleep","3600"]
    volumeMounts:
    - mountPath: /tmp/cores
      name: coredump-path
```

Create a pod using kubectl.

kubectl create -f pod.yaml

Verification

After the pod is created, access the container and trigger a segmentation fault of the current shell terminal.

```
$ kubectl get pod
NAME                       READY   STATUS    RESTARTS   AGE
coredump                   1/1     Running   0          56s
$ kubectl exec -it coredump -- /bin/bash
root@coredump:/# kill -s SIGSEGV $$
command terminated with exit code 139
```

Log in to the node and check whether a core file is generated in the **/home/coredump** directory. The following example indicates that a core file is generated.

```
# ls /home/coredump
core.coredump.bash.18.1650438992
```

12.11 Configuring Parameters to Delay the Pod Startup in a CCE Turbo Cluster

Application Scenarios

In a CCE Turbo cluster, the routing rules of the peer pod may take effect slowly in some specific scenarios like cross-VPC and private line interconnections. To avoid this problem, configure parameters to delay the pod startup.

You can also create enterprise routers to connect to the peer VPCs. For details, see **Connecting a Cluster to the Peer VPC Through an Enterprise Router**.

Constraints

Only the CCE Turbo clusters of the following versions support the configuration of this parameter:

- v1.19: v1.19.16-r40 or later
- v1.21: v1.21.11-r0 or later
- v1.23: v1.23.9-r0 or later
- v1.25: v1.25.4-r0 or later

Using kubectl

You can add annotations to a workload to configure whether to delay the pod startup.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: nginx
      annotations:
        cni.yangtse.io/readiness-delay-seconds: "20"
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 100m
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
      imagePullSecrets:
        - name: default-secret
```

In this example, the Deployment needs to be configured to access the IP addresses of the cross-VPC VMs. The maximum number of replicas of this Deployment is **10** and the maximum rolling upgrade surge is **25%**, which is, the maximum number of pods to be upgraded concurrently is 13. In the annotation, the pod startup delay is set to **20s**. This ensures that the cross-VPC network can be accessed normally after the pod is started.

Table 12-5 Configuring the annotation

Annotation	Default Value	Description	Value Range
cni.yangtse.io/readiness-delay-seconds	None	Indicates the waiting time for the pod health check.	0-60

13 Permission

13.1 Configuring kubeconfig for Fine-Grained Management on Cluster Resources

Application Scenarios

By default, the kubeconfig file provided by CCE for users has permissions bound to the **cluster-admin** role, which are equivalent to the permissions of user **root**. It is difficult to implement refined management on users with such permissions.

Purpose

Cluster resources are managed in a refined manner so that specific users have only certain permissions (such as adding, querying, and modifying resources).

Precautions

Ensure that kubectl is available on your host. If not, download it from **here** (corresponding to the cluster version or the latest version).

Configuration Method

NOTE

In the following example, only pods and Deployments in the **test** space can be viewed and added, and they cannot be deleted.

Step 1 Set the service account name to **my-sa** and namespace to **test**.

kubectl create sa **my-sa** -n **test**

```
[root@test-arm-54016 ~]#
[root@test-arm-54016 ~]# kubectl create sa my-sa -n test
serviceaccount/my-sa created
[root@test-arm-54016 ~]#
```

Step 2 Configure the role table and assign operation permissions to different resources.

vi **role-test.yaml**

The content is as follows:

NOTE

In this example, the permission rules include the read-only permission (get/list/watch) of pods in the **test** namespace, and the read (get/list/watch) and create permissions of deployments.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: myrole
  namespace: test
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - pods
  - deployments
  verbs:
  - get
  - list
  - watch
  - create
```

Create a Role.

```
kubectl create -f role-test.yaml
```

```
[root@test-arm-54016 ~]# kubectl create -f role-test.yaml
role.rbac.authorization.k8s.io/myrole created
[root@test-arm-54016 ~]#
```

Step 3 Create a RoleBinding and bind the service account to the role so that the user can obtain the corresponding permissions.

```
vi myrolebinding.yaml
```

The content is as follows:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: myrolebinding
  namespace: test
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: myrole
subjects:
- kind: ServiceAccount
  name: my-sa
  namespace: test
```

Create a RoleBinding.

```
kubectl create -f myrolebinding.yaml
```



```
[root@test-arm-54016 ~]# kubectl create -f myrolebinding.yaml
rolebinding.rbac.authorization.k8s.io/myrolebinding created
[root@test-arm-54016 ~]#
```

The user information is configured. Now perform **Step 5** to **Step 7** to write the user information to the configuration file.

Step 4 Manually create a token that is valid for a long time for ServiceAccount.

```
vi my-sa-token.yaml
```

The content is as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-sa-token-secret
  namespace: test
  annotations:
    kubernetes.io/service-account.name: my-sa
type: kubernetes.io/service-account-token
```

Create a token:

```
kubectl create -f my-sa-token.yaml
```

Step 5 Configure the cluster information.

1. Decrypt the **ca.crt** file in the secret and export it.

```
kubectl get secret my-sa-token-secret -n test -oyaml |grep ca.crt: | awk '{print $2}' |base64 -d > /home/ca.crt
```

2. Set a cluster access mode. **test-arm** specifies the cluster to be accessed. **https://192.168.0.110:5443** specifies the apiserver IP address of the cluster. For details about how to obtain the IP address, see **Figure 13-1**. **/home/test.config** specifies the path for storing the configuration file.
   - If the internal API server address is used, run the following command:

     ```
     kubectl config set-cluster test-arm --server=https://192.168.0.110:5443  --certificate-authority=/home/ca.crt  --embed-certs=true --kubeconfig=/home/test.config
     ```

   - If the public API server address is used, run the following command:

     ```
     kubectl config set-cluster test-arm --server=https://192.168.0.110:5443 --kubeconfig=/home/test.config --insecure-skip-tls-verify=true
     ```

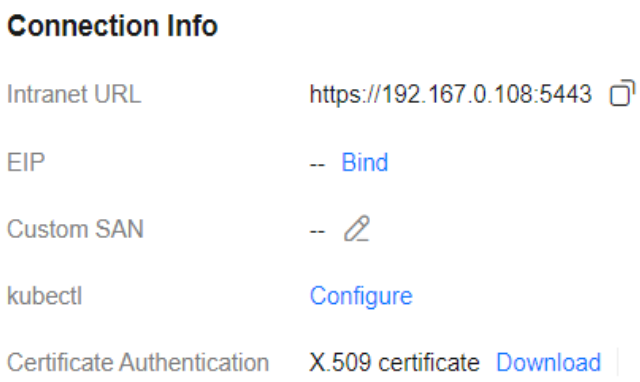
```
[root@test-arm-54016 home]# kubectl config set-cluster test-arm --server=https://10.0.1.100:5443  --certificate-authority=/home/ca.crt  --embed-certs=true --kubeconfig=/home/test.config
Cluster "test-arm" set.
[root@test-arm-54016 home]# _
```

NOTE

If you **perform operations on a node in the cluster** or **the node that uses the configuration is a cluster node**, do not set the path of kubeconfig to **/root/.kube/config**.

By default, the apiserver IP address of the cluster is a private IP address. After an EIP is bound, you can use the public network IP address to access the apiserver.

Figure 13-1 Obtaining the internal or public API server address



Step 6 Configure the cluster authentication information.

1. Obtain the cluster token. (If the token is obtained in GET mode, run **based64 -d** to decode the token.)

```
token=$(kubectl describe secret my-sa-token-secret -n test | awk '/token:/{print $2}')
```

2. Set the cluster user **ui-admin**.

```
kubectl config set-credentials ui-admin --token=$token --kubeconfig=/home/test.config
```

```
[root@test-arm-54016 home]# kubectl config set-credentials ui-admin --token=$token --kubeconfig=/home/test.config
User "ui-admin" set.
[root@test-arm-54016 home]#
```

Step 7 Configure the context information for cluster authentication access. **ui-admin@test** specifies the context name.

```
kubectl config set-context ui-admin@test --cluster=test-arm --user=ui-admin --kubeconfig=/home/test.config
```

```
[root@test-arm-54016 home]# kubectl config set-context ui-admin@test --cluster=test-arm --user=ui-admin --kubeconfig=/home/test.
config
Context "ui-admin@test" created.
[root@test-arm-54016 home]#
```

Step 8 Configure the context. For details about how to use the context, see **Verification**.

```
kubectl config use-context ui-admin@test --kubeconfig=/home/test.config
```

```
[paas@test-arm-54016 home]$ kubectl config use-context ui-admin@test --kubeconfig=/home/test.config
Switched to context "ui-admin@test".
[paas@test-arm-54016 home]$
```

NOTE

If you want to assign other users the above permissions to perform operations on the cluster, provide the generated configuration file **/home/test.config** to the user after performing step **Step 7**. The user must ensure that the host can access the API server address of the cluster. When performing step **Step 8** on the host and using kubectl, the user must set the kubeconfig parameter to the path of the configuration file.

----End

Verification

1. Pods in the **test** namespace cannot access pods in other namespaces.

```
kubectl get pod -n test --kubeconfig=/home/test.config
```

```
[paas@test-arm-54016 home]$ kubectl get pod -n test --kubeconfig=/home/test.config
NAME                        READY   STATUS             RESTARTS   AGE
test-pod-56cfcbf45b-l2q92   0/1     CrashLoopBackOff   27         91m
[paas@test-arm-54016 home]$
[paas@test-arm-54016 home]$ kubectl get pod --kubeconfig=/home/test.config
Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:test:my-sa" cannot list resource "pods" in API gro
up "" in the namespace "default"
[paas@test-arm-54016 home]$
```


2. Pods in the **test** namespace cannot be deleted.

```
[paas@test-arm-54016 home]$ kubectl delete pod -n test test-pod-56cfcbf45b-12q92 --kubeconfig=/home/test.config
Error from server (Forbidden): pods "test-pod-56cfcbf45b-12q92" is forbidden: User "system:serviceaccount:test:my-sa" cannot del
ete resource "pods" in API group "" in the namespace "test"
[paas@test-arm-54016 home]$
```

Further Readings

For more information about users and identity authentication in Kubernetes, see **Authenticating**.

13.2 Performing Cluster Namespace RBAC

Background

CCE permissions are classified into cluster permissions and namespace permissions. Namespace permissions are based on Kubernetes RBAC and can be used to grant permissions on resources in clusters and namespaces.

Currently, the CCE console provides four types of namespace-level ClusterRole permissions by default: cluster-admin, admin, edit, and view. However, these permissions apply to resources in the namespace regardless of resource types (pods, Deployments, and Services).

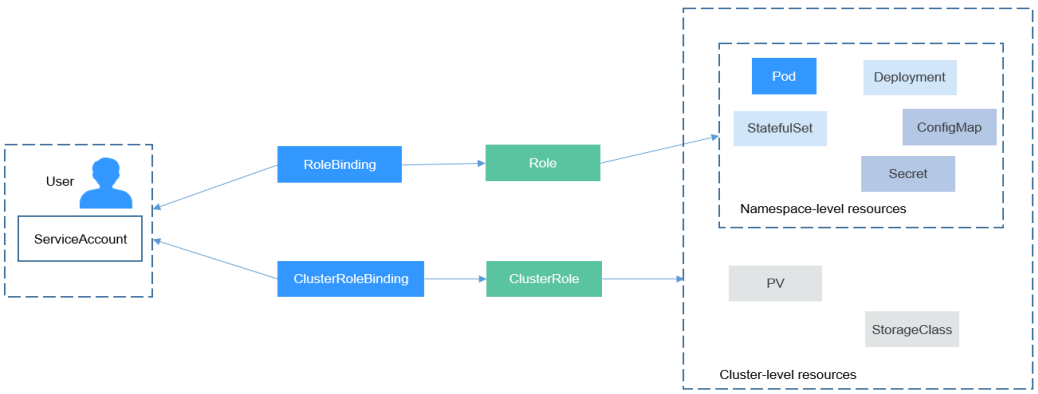
Solution

Kubernetes RBAC enables you to easily grant permissions on namespace resources.

- Role: defines a set of rules for accessing Kubernetes resources in a namespace.
- RoleBinding: defines the relationship between users and roles.
- ClusterRole: defines a set of rules for accessing Kubernetes resources in a cluster (including all namespaces).
- ClusterRoleBinding: defines the relationship between users and cluster roles.

Role and ClusterRole specify actions that can be performed on specific resources. RoleBinding and ClusterRoleBinding bind roles to specific users, user groups, or ServiceAccounts. See the following figure.

Figure 13-2 Role binding



The user in the preceding figure can be an IAM user or user group in CCE. You can efficiently control permissions on namespace resources through RoleBindings.

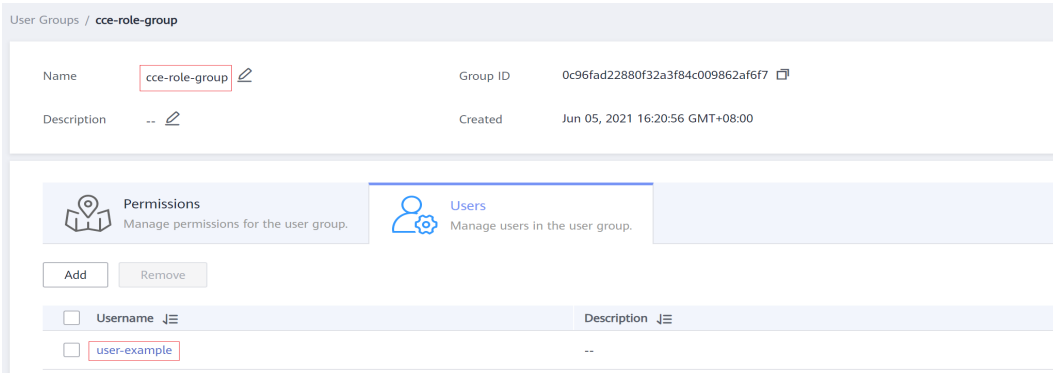
The section describes how to use Kubernetes RBAC to grant user **user-example** with the permission for viewing pods. (This is the only permission the user has.)

Prerequisites

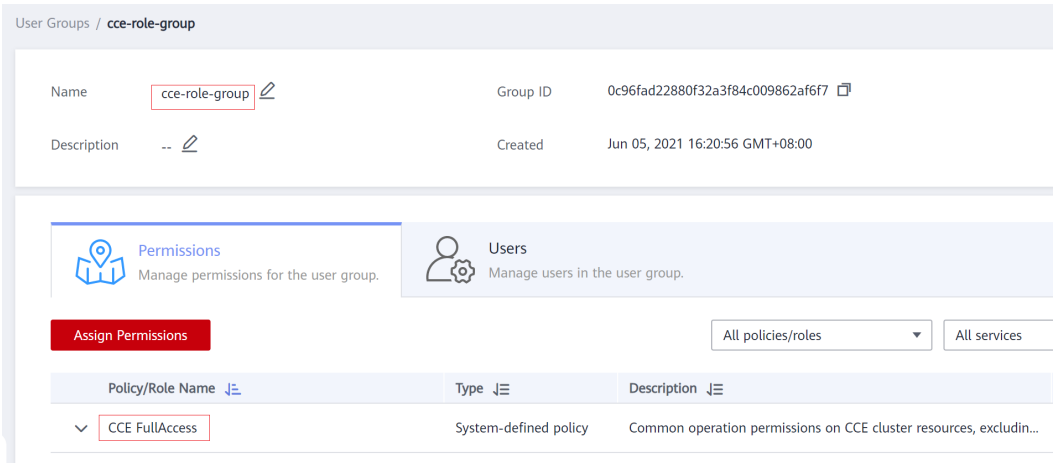
RBAC is supported only on clusters of v1.11.7-r2 or later.

Creating an IAM User and User Group

Log in to IAM and create an IAM user named **user-example** and a user group named **cce-role-group**. For details about how to create an IAM user and user group, see **Creating IAM Users** and **Creating User Groups**.



Grant the **CCE FullAccess** permission to the **cce-role-group** user group. For details about how to grant permissions to a user group, see **Assigning Permissions to User Groups**.



CCE FullAccess has the permissions for cluster operations (such as cluster creation), but does not have the permissions to operate Kubernetes resources (such as viewing pods).

Creating a Cluster

Log in to CCE and create a cluster.

NOTICE

Do not use IAM user **user-example** to create a cluster because CCE automatically assigns the cluster-admin permissions of all namespaces in the cluster to the user who creates the cluster. That is, the user can fully control the resources in the cluster and all its namespaces.

Log in to the CCE console as IAM user **user-example**, **download the kubectl configuration file in the cluster and connect to the cluster**. Run the following command to obtain the pod information. The output shows that **user-example** does not have the permission to view the pods or other resources. This indicates that **user-example** does not have the permissions to operate Kubernetes resources.

```
# kubectl get pod
Error from server (Forbidden): pods is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "pods" in API group "" in the namespace "default"
# kubectl get deploy
Error from server (Forbidden): deployments.apps is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list resource "deployments" in API group "apps" in the namespace "default"
```

Creating a Role and RoleBinding

Log in to the CCE console, **download the kubectl configuration file in the cluster and connect to the cluster**. Create a Role and RoleBinding.

NOTE

Log in as the account used to create the cluster because CCE automatically assigns the cluster-admin permissions to the account, which means that the account has the permissions to create Roles and RoleBindings. Alternatively, you can use IAM users who have the permissions to create Roles and RoleBindings.

The procedure for creating a Role is very simple. To be specific, specify a namespace and then define rules. The rules in the following example are to allow GET and LIST operations on pods in the default namespace.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default                    # Namespace
  name: role-example
rules:
- apiGroups: [""]
  resources: ["pods"]                   # The pod can be accessed.
  verbs: ["get", "list"]                # The GET and LIST operations can be performed.
```

- **apiGroups** indicates the API group to which the resource belongs.
- **resources** indicates the resources that can be operated. Pods, Deployments, ConfigMaps, and other Kubernetes resources are supported.
- **verbs** indicates the operations that can be performed. **get** indicates querying a specific object, and **list** indicates listing all objects of a certain type. Other value options include **create**, **update**, and **delete**.

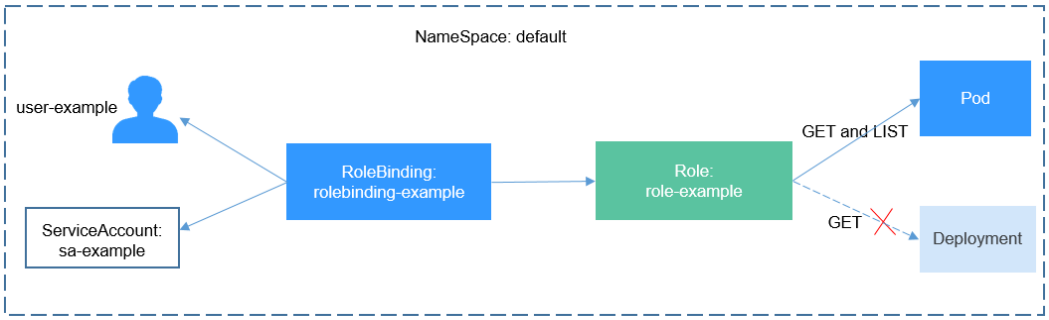
For details, see **Using RBAC Authorization**.

After creating a Role, you can bind the Role to a specific user, which is called RoleBinding. The following shows an example:


```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: RoleBinding-example
  namespace: default
  annotations:
    CCE.com/IAM: 'true'
roleRef:
  kind: Role
  name: role-example
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: User
  name: 0c97ac3cb280f4d91fa7c0096739e1f8    # IAM user ID
  apiGroup: rbac.authorization.k8s.io
```

The **subjects** section binds a Role with an IAM user so that the IAM user can obtain the permissions defined in the Role, as shown in the following figure.

Figure 13-3 Binding a role to a user



You can also specify a user group in the **subjects** section. In this case, all users in the user group obtain the permissions defined in the Role.

```
subjects:
- kind: Group
  name: 0c96fad22880f32a3f84c009862af6f7    # User group ID
  apiGroup: rbac.authorization.k8s.io
```

Verification

Use IAM user **user-example** to connect to the cluster and view the pods. The pods can be viewed.

```
# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
nginx-658dff48ff-7rkph              1/1     Running   0          4d9h
nginx-658dff48ff-njdhj              1/1     Running   0          4d9h
# kubectl get pod nginx-658dff48ff-7rkph
NAME                    READY   STATUS    RESTARTS   AGE
nginx-658dff48ff-7rkph  1/1     Running   0          4d9h
```

Try querying Deployments and Services in the namespace. The output shows **user-example** does not have the corresponding permissions. Try querying the pods in namespace kube-system. The output shows **user-example** does not have the corresponding permission, either. This indicates that the IAM user **user-example** has only the GET and LIST Pod permissions in the default namespace, which is the same as expected.

```
# kubectl get deploy
Error from server (Forbidden): deployments.apps is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8"
```



```
cannot list resource "deployments" in API group "apps" in the namespace "default"
# kubectl get svc
Error from server (Forbidden): services is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list
resource "services" in API group "" in the namespace "default"
# kubectl get pod --namespace=kube-system
Error from server (Forbidden): pods is forbidden: User "0c97ac3cb280f4d91fa7c0096739e1f8" cannot list
resource "pods" in API group "" in the namespace "kube-system"
```


14 Release

14.1 Overview

Background

When switching between old and new services, you may be challenged in ensuring the system service continuity. If a new service version is directly released to all users at a time, it can be risky because once an online accident or bug occurs, the impact on users is great. It could take a long time to fix the issue. Sometimes, the version has to be rolled back, which severely affects user experience.

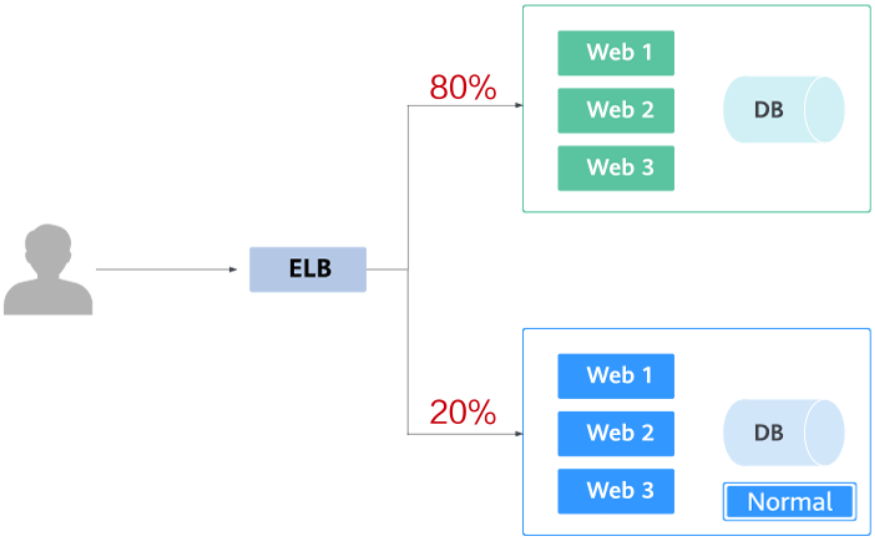
Solution

Several release policies are developed for service upgrade: grayscale release, blue-green deployment, A/B testing, rolling upgrade, and batch suspension of release. Traffic loss or service unavailability caused by releases can be avoided as much as possible.

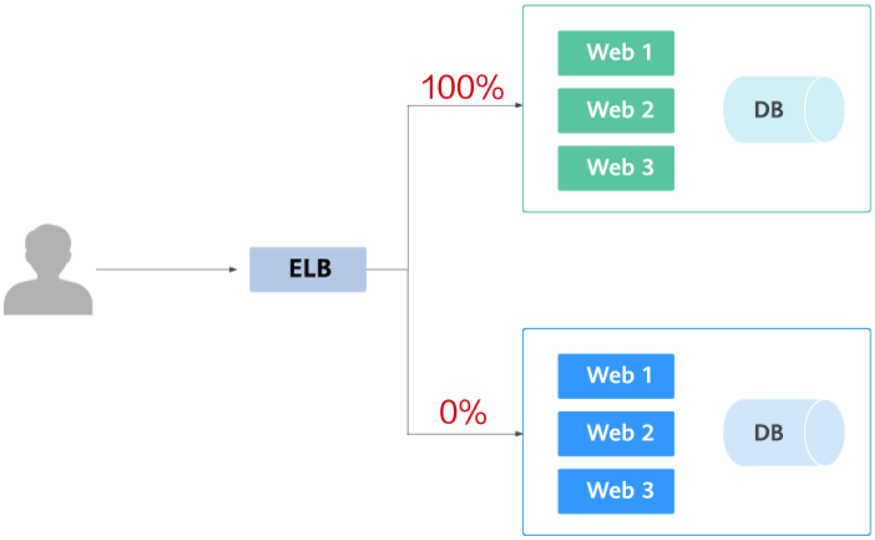
This document describes the principles and practices of grayscale release and blue-green deployment.

- Grayscale release, also called canary release, is a smooth iteration mode for version upgrade. During the upgrade, some users use the new version, while other users continue to use the old version. After the new version is stable and ready, it gradually takes over all the live traffic. In this way, service risks brought by the release of the new version can be minimized, the impact of faults can be reduced, and quick rollback is supported.

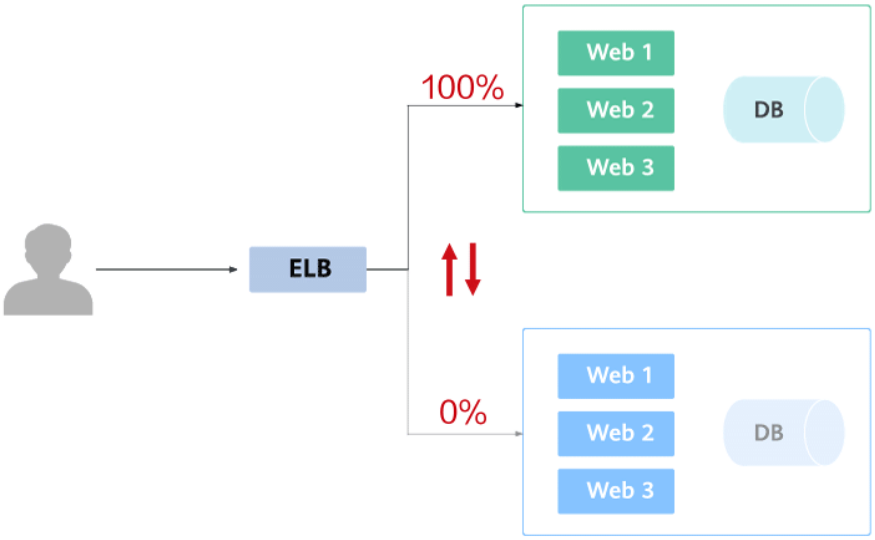
  The following figure shows the general process of grayscale release. First, divide 20% of all service traffic to the new version. If the service version runs normally, gradually increase the traffic proportion and continue to test the performance of the new version. If the new version is stable, switch all traffic to it and bring the old version offline.



If an exception occurs in the new version when 20% of the traffic goes to the new version, you can quickly switch back to the old version.



- Blue-green deployment provides a zero-downtime, predictable manner for releasing applications to reduce service interruption during the release. A new version is deployed while the old version is retained. The two versions are online at the same time. The new and old versions work in hot backup mode. The route weight is switched (0 or 100) to enable different versions to go online or offline. If a problem occurs, the version can be quickly rolled back.



Realizing Grayscale Release or Blue-Green Deployment

Kubernetes-native features can be used to implement simple grayscale release or blue-green deployment. For example, you can change the value of the label that determines the service version in the selector of a Service to change the pod backing the Service. In this way, the service can be directly switched from one version to another. If you have complex grayscale release or blue-green deployment requirements, you can deploy service meshes and open-source tools, such as Nginx Ingress and Traefik, to the cluster. You can obtain detailed instructions in the following sections:

- **Using Services to Implement Simple Grayscale Release and Blue-Green Deployment**
- **Using Nginx Ingress to Implement Grayscale Release and Blue-Green Deployment**

Table 14-1 Implementation mode comparison

Implementation	Application Scenario	Feature	Disadvantage
Service	Simple release requirements and small-scale test scenarios	No need to introduce too many plug-ins or complex configurations	Manual operations and poor automation

Implementation	Application Scenario	Feature	Disadvantage
Nginx Ingress	No special requirements.	• Only the annotation supported by Nginx Ingress needs to be configured. • Header-based, cookie-based, and service weight-based traffic division policies are supported.	The nginx-ingress add-on needs to be installed in the cluster, which consumes resources.

Both Services and Nginx Ingresses use open source Kubernetes capabilities to implement grayscale release and blue-green deployment. In this process, CCE allows you to easily perform the following operations:

- All resources can be created, viewed, and modified on the console, which is more intuitive than the kubectl command line tool.
- LoadBalancer Services are supported by the ELB service. When creating a Service, you can use an existing ELB instance or create a new one.
- The nginx-ingress add-on can be installed in just a few clicks, and ELB load balancers can be created and interconnected during the installation.

14.2 Using Services to Implement Simple Grayscale Release and Blue-Green Deployment

To implement grayscale release for a CCE cluster, deploy other open-source tools, such as Nginx Ingress, to the cluster or deploy services to a service mesh. These solutions are difficult to implement. If your grayscale release requirements are simple and you do not want to introduce too many plug-ins or complex configurations, you can refer to this section to implement simple grayscale release and blue-green deployment based on native Kubernetes features.

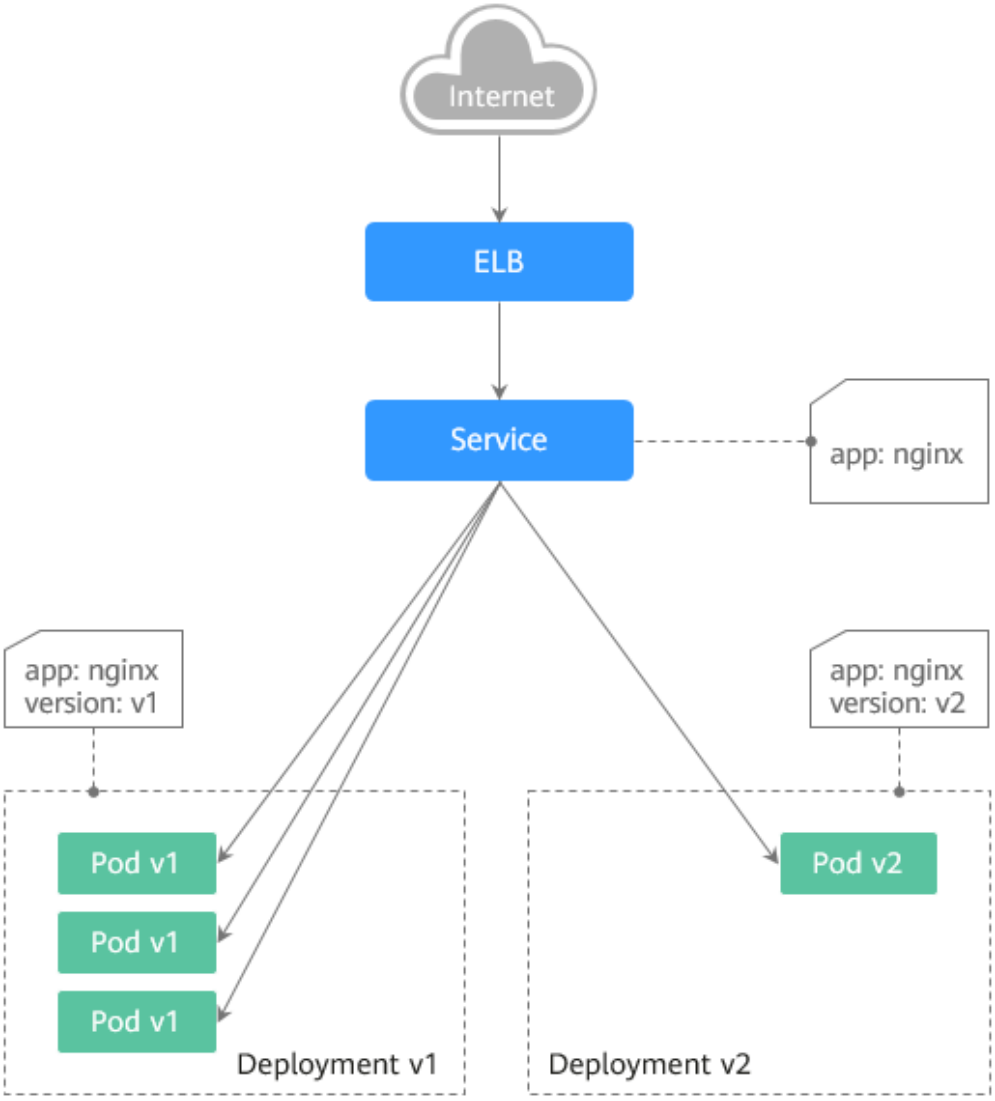
Principles

Users usually use Kubernetes objects such as Deployments and StatefulSets to deploy services. Each workload manages a group of pods. The following figure uses Deployment as an example.

Generally, a Service is created for each workload. The Service uses the selector to match the backend pod. Other Services or objects outside the cluster can access the pods backing the Service. If a pod needs to be exposed, set the Service type to LoadBalancer. The ELB load balancer functions as the traffic entrance.

- **Grayscale release principles**

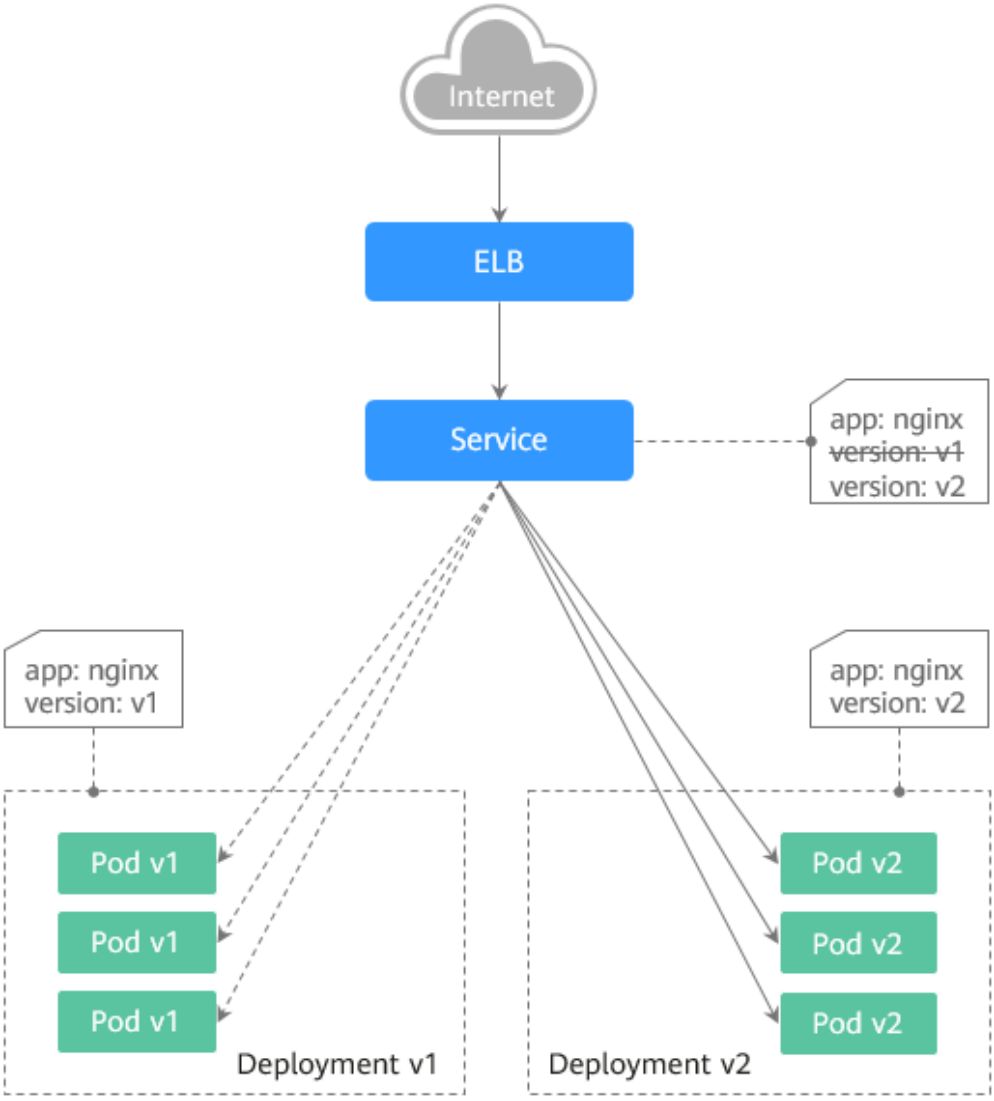
  Take a Deployment as an example. A Service, in most cases, will be created for each Deployment. However, Kubernetes does not require that Services and Deployments correspond to each other. A Service uses a selector to match backend pods. If pods of different Deployments are selected by the same selector, a Service corresponds to multiple versions of Deployments. You can adjust the number of replicas of Deployments of different versions to adjust the weights of services of different versions to achieve grayscale release. The following figure shows the process:



- **Blue-green deployment principles**

  Take a Deployment as an example. Two Deployments of different versions have been deployed in the cluster, and their pods are labeled with the same key but different values to distinguish versions. A Service uses the selector to select the pod of a Deployment of a version. In this case, you can change the

value of the label that determines the version in the Service selector to change the pod backing the Service. In this way, you can directly switch the service traffic from one version to another. The following figure shows the process:



Prerequisites

The Nginx image has been uploaded to SWR. The Nginx images have two versions: v1 and v2. The welcome pages are **Nginx-v1** and **Nginx-v2**.

Resource Creation

You can use YAML to deploy Deployments and Services in either of the following ways:

- On the **Create Deployment** page, click **Create YAML** on the right and edit the YAML file in the window.
- Save the sample YAML file in this section as a file and use kubectl to specify the YAML file. For example, run the **kubectl create -f xxx.yaml** command.

Step 1: Deploy Services of Two Versions

Two versions of Nginx services are deployed in the cluster to provide external access through ELB.

Step 1 Create a Deployment of the first version. The following uses nginx-v1 as an example. Example YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v1
spec:
  replicas: 2             # Number of replicas of the Deployment, that is, the number of pods
  selector:               # Label selector
    matchLabels:
      app: nginx
      version: v1
  template:
    metadata:
      labels:             # Pod label
        app: nginx
        version: v1
    spec:
      containers:
      - image: {your_repository}/nginx:v1 # The image used by the container is nginx:v1.
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```

Step 2 Create a Deployment of the second version. The following uses nginx-v2 as an example. Example YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-v2
spec:
  replicas: 2             # Number of replicas of the Deployment, that is, the number of pods
  selector:               # Label selector
    matchLabels:
      app: nginx
      version: v2
  template:
    metadata:
      labels:             # Pod label
        app: nginx
        version: v2
    spec:
      containers:
      - image: {your_repository}/nginx:v2   # The image used by the container is nginx:v2.
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```


You can log in to the CCE console to view the deployment status.

----End

Step 2: Implement Grayscale Release

Step 1 Create a LoadBalancer Service for the Deployment. Do not specify the version in the selector. Enable the Service to select the pods of the Deployments of two versions. Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: 586c97da-a47c-467c-a615-bd25a20de39c    # ID of the ELB load balancer. Replace it with the actual value.
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:            # The selector does not contain version information.
    app: nginx
  type: LoadBalancer   # Service type (LoadBalancer)
```

Step 2 Run the following command to test the access:

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

<EXTERNAL_IP> indicates the IP address of the ELB load balancer.

The command output is as follows (Half of the responses are from the Deployment of version v1, and the other half are from version v2):

```
Nginx-v2
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v2
Nginx-v2
```

Step 3 Use the console or kubectl to adjust the number of replicas of the Deployments. Change the number of replicas to 4 for v1 and 1 for v2.

kubectl scale deployment/nginx-v1 --replicas=4

kubectl scale deployment/nginx-v2 --replicas=1

Step 4 Run the following command to test the access again:

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

<EXTERNAL_IP> indicates the IP address of the ELB load balancer.

In the command output, among the 10 access requests, only two responses are from the v2 version. The response ratio of the v1 and v2 versions is the same as the ratio of the number of replicas of the v1 and v2 versions, that is, 4:1. Grayscale release is implemented by controlling the number of replicas of services of different versions.


```
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v2
Nginx-v1
Nginx-v1
Nginx-v1
```

NOTE

If the ratio of v1 to v2 is not 4:1, you can set the number of access times to a larger value, for example, 20. Theoretically, the more the times, the closer the response ratio between v1 and v2 is to 4:1.

----End

Step 3: Implement Blue-Green Deployment

Step 1 Create a LoadBalancer Service for a deployed Deployment and specify that the v1 version is used. Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    kubernetes.io/elb.id: 586c97da-a47c-467c-a615-bd25a20de39c    # ID of the ELB load balancer. Replace it with the actual value.
  name: nginx
spec:
  ports:
  - name: service0
    port: 80
    protocol: TCP
    targetPort: 80
  selector:            # Set the version to v1 in the selector.
    app: nginx
    version: v1
  type: LoadBalancer    # Service type (LoadBalancer)
```

Step 2 Run the following command to test the access:

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

<EXTERNAL_IP> indicates the IP address of the ELB load balancer.

The command output is as follows (all responses are from the v1 version):

```
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
Nginx-v1
```

Step 3 Use the console or kubectl to modify the selector of the Service so that the v2 version is selected.

kubectl patch service nginx -p '{"spec":{"selector":{"version":"v2"}}}'

Step 4 Run the following command to test the access again:

for i in {1..10}; do curl <EXTERNAL_IP>**; done;**

<EXTERNAL_IP> indicates the IP address of the ELB load balancer.

The returned results show that are all responses are from the v2 version. The blue-green deployment is successfully implemented.

```
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
Nginx-v2
```

----End

14.3 Using Nginx Ingress to Implement Grayscale Release and Blue-Green Deployment

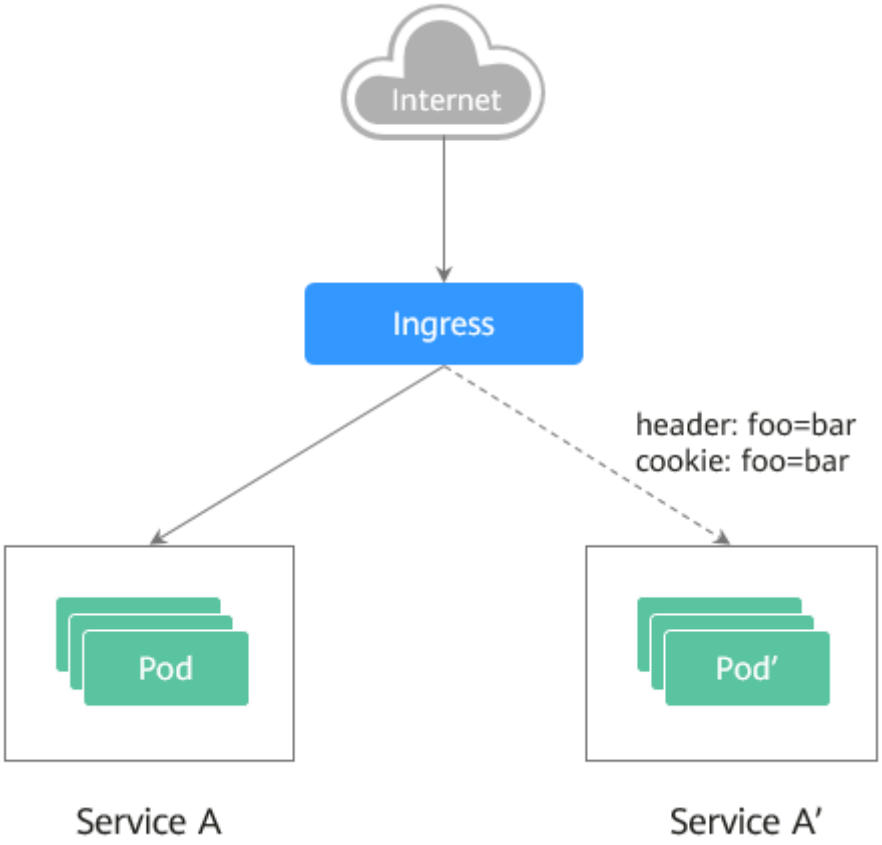
This section describes the scenarios and practices of using Nginx Ingress to implement grayscale release and blue-green deployment.

Application Scenarios

Nginx Ingress supports three traffic division policies based on the header, cookie, and service weight. Based on these policies, the following two release scenarios can be implemented:

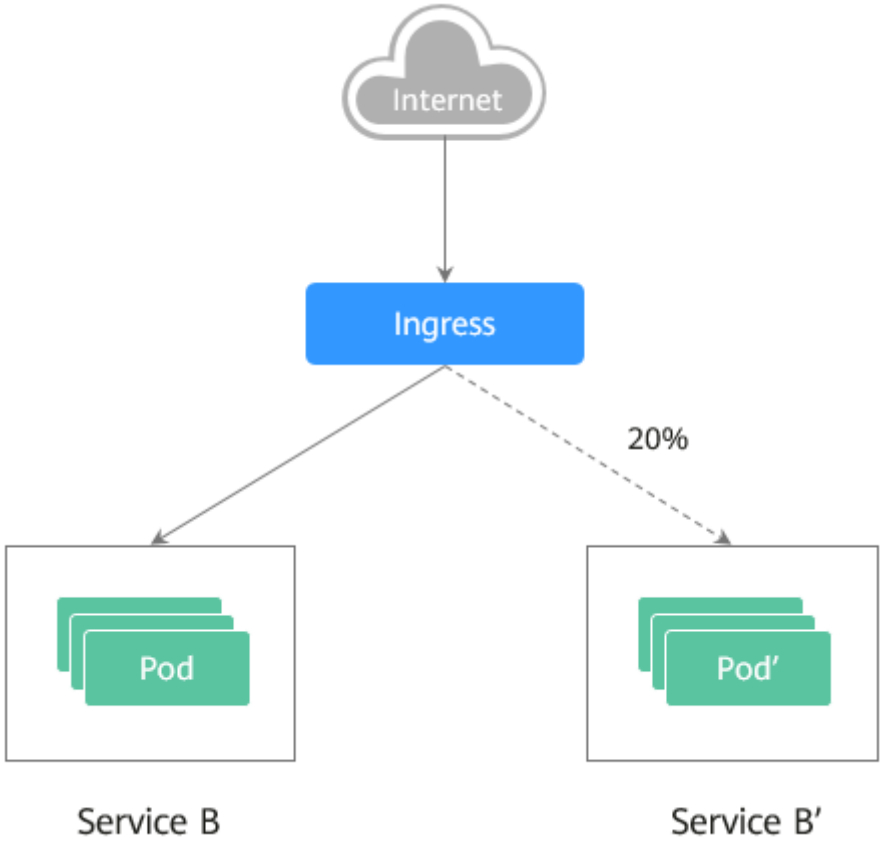
- **Scenario 1: Split some user traffic to the new version.**

  Assume that Service A that provides layer-7 networking is running. A new version is ready to go online, but you do not want to replace the original Service A. You want to forward the user requests whose header or cookie contains **foo=bar** to the new version of Service A. After the new version runs stably for a period of time, you can gradually bring the new version online and smoothly bring the old version offline. The following figure shows the process:



- **Scenario 2: Split a certain proportion of traffic to the new version.**

  Assume that Service B that provides layer-7 services is running. After some problems are resolved, a new version of Service B needs to be released. However, you do not want to replace the original Service B. Instead, you want to switch 20% traffic to the new version of Service B. After the new version runs stably for a period of time, you can switch all traffic from the old version to the new version and smoothly bring the old version offline.



Annotations

Nginx Ingress supports release and testing in different scenarios by configuring annotations for grayscale release, blue-green deployment, and A/B testing. The implementation process is as follows: Create two ingresses for the service. One is a common ingress, and the other is an ingress with the annotation **nginx.ingress.kubernetes.io/canary: "true"**, which is called a canary ingress. Configure a traffic division policy for the canary ingress. The two ingresses cooperate with each other to implement release and testing in multiple scenarios. The annotation of Nginx Ingress supports the following rules:

- **nginx.ingress.kubernetes.io/canary-by-header**

  Header-based traffic division, which is applicable to grayscale release. If the request header contains the specified header name and the value is **always**, the request is forwarded to the backend service defined by the canary ingress. If the value is **never**, the request is not forwarded and a rollback to the source version can be performed. If other values are used, the annotation is ignored and the request traffic is allocated according to other rules based on the priority.

- **nginx.ingress.kubernetes.io/canary-by-header-value**

  This rule must be used together with canary-by-header. You can customize the value of the request header, including but not limited to **always** or **never**. If the value of the request header matches the specified custom value, the request is forwarded to the corresponding backend service defined by the canary ingress. If the values do not match, the annotation is ignored and the request traffic is allocated according to other rules based on the priority.

- **nginx.ingress.kubernetes.io/canary-by-header-pattern**

  This rule is similar to canary-by-header-value. The only difference is that this annotation uses a regular expression, not a fixed value, to match the value of the request header. If this annotation and canary-by-header-value exist at the same time, this one will be ignored.

- **nginx.ingress.kubernetes.io/canary-by-cookie**

  Cookie-based traffic division, which is applicable to grayscale release. Similar to canary-by-header, this annotation is used for cookies. Only **always** and **never** are supported, and the value cannot be customized.

- **nginx.ingress.kubernetes.io/canary-weight**

  Traffic is divided based on service weights, which is applicable to blue-green deployment. This annotation indicates the percentage of traffic allocated by the canary ingress. The value ranges from 0 to 100. For example, if the value is set to **100**, all traffic is forwarded to the backend service backing the canary ingress.

NOTE

- The preceding annotation rules are evaluated based on the priority. The priority is as follows: canary-by-header -> canary-by-cookie -> canary-weight.
- When an ingress is marked as a canary ingress, all non-canary annotations except **nginx.ingress.kubernetes.io/load-balance** and **nginx.ingress.kubernetes.io/upstream-hash-by** are ignored.
- For more information, see **Annotations**.

Prerequisites

- To use Nginx Ingress to implement grayscale release of a cluster, install the nginx-ingress add-on as the Ingress Controller and expose a unified traffic entrance externally. For details, see **Installing the Add-on**.
- The Nginx image has been uploaded to SWR. The Nginx images have two versions. The welcome pages are **Old Nginx** and **New Nginx**.

Resource Creation

You can use YAML to deploy Deployments and Services in either of the following ways:

- On the **Create Deployment** page, click **Create YAML** on the right and edit the YAML file in the window.
- Save the sample YAML file in this section as a file and use kubectl to specify the YAML file. For example, run the **kubectl create -f xxx.yaml** command.

Step 1: Deploy Services of Two Versions

Two versions of Nginx are deployed in the cluster, and Nginx Ingress is used to provide layer-7 domain name access for external systems.

Step 1 Create a Deployment and Service for the first version. This section uses old-nginx as an example. Example YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```



```
  name: old-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: old-nginx
  template:
    metadata:
      labels:
        app: old-nginx
    spec:
      containers:
      - image: {your_repository}/nginx:old  # The image used by the container is nginx:old.
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret

---

apiVersion: v1
kind: Service
metadata:
  name: old-nginx
spec:
  selector:
    app: old-nginx
  ports:
  - name: service0
    targetPort: 80
    port: 8080
    protocol: TCP
  type: NodePort
```

Step 2 Create a Deployment and Service for the second version. This section uses new-nginx as an example. Example YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: new-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: new-nginx
  template:
    metadata:
      labels:
        app: new-nginx
    spec:
      containers:
      - image: {your_repository}/nginx:new  # The image used by the container is nginx:new.
        name: container-0
        resources:
          limits:
            cpu: 100m
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      imagePullSecrets:
      - name: default-secret
```



```
---

apiVersion: v1
kind: Service
metadata:
  name: new-nginx
spec:
  selector:
    app: new-nginx
  ports:
  - name: service0
    targetPort: 80
    port: 8080
    protocol: TCP
  type: NodePort
```

You can log in to the CCE console to view the deployment status.

Step 3 Create an ingress to expose the service and point to the service of the old version. Example YAML:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: gray-release
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx # Use the Nginx ingress.
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: old-nginx      # Specify old-nginx as the backend service.
              servicePort: 80
```

Step 4 Run the following command to verify the access:

```
curl -H "Host: www.example.com"  http://<EXTERNAL_IP>
```

In the preceding command, <EXTERNAL_IP> indicates the external IP address of the Nginx ingress.

Expected outputs:

```
Old Nginx
```

----End

Step 2: Launch the New Version of the Service in Grayscale Release Mode

Set the traffic division policy for the service of the new version. CCE supports the following policies for grayscale release and blue-green deployment:

Header-based, **cookie-based**, and **weight-based** traffic division rules

Grayscale release can be implemented based on all these policies. Blue-green deployment can be implemented by adjusting the new service weight to 100%. For details, see the following examples.

CAUTION

Pay attention to the following:

- Only one canary ingress can be defined for the same service so that the backend service supports a maximum of two versions.
- Even if the traffic is completely switched to the canary ingress, the old version service must still exist. Otherwise, an error is reported.

- **Header-based rules**

  In the following example, only the request whose header contains **Region** set to **bj** or **gz** can be forwarded to the service of the new version.

  a. Create a canary ingress, set the backend service to the one of the new versions, and add annotations.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"                    # Enable canary.
    nginx.ingress.kubernetes.io/canary-by-header: "Region"
    nginx.ingress.kubernetes.io/canary-by-header-pattern: "bj|gz"    # Requests whose header
contains Region with the value bj or gz are forwarded to the canary ingress.
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx      # Specify new-nginx as the backend service.
              servicePort: 80
```

  b. Run the following command to test the access:

```
$ curl -H "Host: www.example.com" -H "Region: bj" http://<EXTERNAL_IP>
New Nginx
$ curl -H "Host: www.example.com" -H "Region: sh" http://<EXTERNAL_IP>
Old Nginx
$ curl -H "Host: www.example.com" -H "Region: gz" http://<EXTERNAL_IP>
New Nginx
$ curl -H "Host: www.example.com" http://<EXTERNAL_IP>
Old Nginx
```

  In the preceding command, <EXTERNAL_IP> indicates the external IP address of the Nginx ingress.

  Only requests whose header contains **Region** with the value **bj** or **gz** are responded by the service of the new version.

- **Cookie-based rules**

  In the following example, only the request whose cookie contains **user_from_bj** can be forwarded to the service of the new version.

  a. Create a canary ingress, set the backend service to the one of the new versions, and add annotations.

  NOTE

  If you have created a canary ingress in the preceding steps, delete it and then perform this step to create a canary ingress.


```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"                       # Enable canary.
    nginx.ingress.kubernetes.io/canary-by-cookie: "user_from_bj"     # Requests whose cookie
contains user_from_bj are forwarded to the canary ingress.
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx      # Specify new-nginx as the backend service.
              servicePort: 80
```

b. Run the following command to test the access:

```
$ curl -s -H "Host: www.example.com" --cookie "user_from_bj=always" http://
<EXTERNAL_IP>
New Nginx
$ curl -s -H "Host: www.example.com" --cookie "user_from_gz=always" http://
<EXTERNAL_IP>
Old Nginx
$ curl -s -H "Host: www.example.com" http://<EXTERNAL_IP>
Old Nginx
```

In the preceding command, <EXTERNAL_IP> indicates the external IP address of the Nginx ingress.

Only requests whose cookie contains **user_from_bj** with the value **always** are responded by the service of the new version.

- **Service weight-based rules**

  Example 1: Only 20% of the traffic is allowed to be forwarded to the service of the new version to implement grayscale release.

  a. Create a canary ingress and add annotations to import 20% of the traffic to the backend service of the new version.

  > **NOTE**
  >
  > If you have created a canary ingress in the preceding steps, delete it and then perform this step to create a canary ingress.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"          # Enable canary.
    nginx.ingress.kubernetes.io/canary-weight: "20"     # Forward 20% of the traffic to the canary
ingress.
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx      # Specify new-nginx as the backend service.
              servicePort: 80
```


b. Run the following command to test the access:

```
$ for i in {1..20}; do curl -H "Host: www.example.com" http://<EXTERNAL_IP>; done;
Old Nginx
Old Nginx
Old Nginx
New Nginx
Old Nginx
New Nginx
Old Nginx
New Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
New Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
Old Nginx
```

In the preceding command, <EXTERNAL_IP> indicates the external IP address of the Nginx ingress.

It can be seen that there is a 4/20 probability that the service of the new version responds, which complies with the setting of the service weight of 20%.

NOTE

After traffic is divided based on the weight (20%), the probability of accessing the new version is close to 20%. The traffic ratio may fluctuate within a small range, which is normal.

Example 2: Allow all traffic to be forwarded to the service of the new version to implement blue-green deployment.

a. Create a canary ingress and add annotations to import 100% of the traffic to the backend service of the new version.

NOTE

If you have created a canary ingress in the preceding steps, delete it and then perform this step to create a canary ingress.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/canary: "true"          # Enable canary.
    nginx.ingress.kubernetes.io/canary-weight: "100"    # All traffic is forwarded to the canary ingress.
    kubernetes.io/elb.port: '80'
spec:
  rules:
    - host: www.example.com
      http:
        paths:
          - path: '/'
            backend:
              serviceName: new-nginx      # Specify new-nginx as the backend service.
              servicePort: 80
```


b. Run the following command to test the access:

```
$ for i in {1..10}; do curl -H "Host: www.example.com" http://<EXTERNAL_IP>; done;
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
New Nginx
```

In the preceding command, <EXTERNAL_IP> indicates the external IP address of the Nginx ingress.

All access requests are responded by the service of the new version, and the blue-green deployment is successfully implemented.

15 Batch Computing

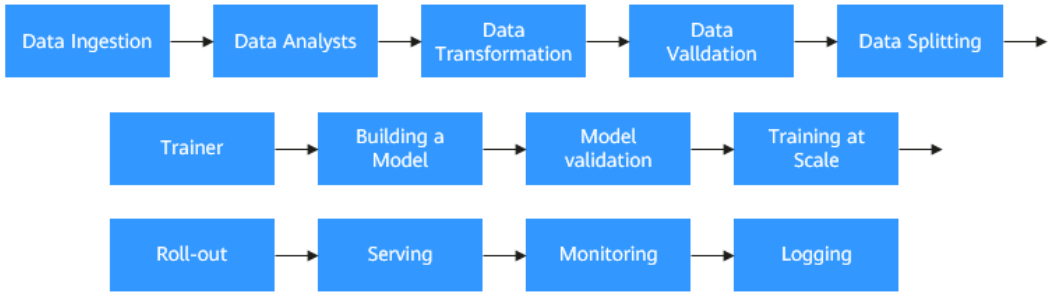
15.1 Running Kubeflow in CCE

15.1.1 Deploying Kubeflow

Background

Building an end-to-end AI computing platform based on Kubernetes is complex. More than a dozen of phases is required. Apart from the familiar model training phase, the process also includes data collection, preprocessing, resource management, feature extraction, data verification, model management, model release, and monitoring. If AI algorithm engineers want to run a model training task, they have to build an entire AI computing platform first. Imagine how time- and labor-consuming that is and how much knowledge and experience it requires.

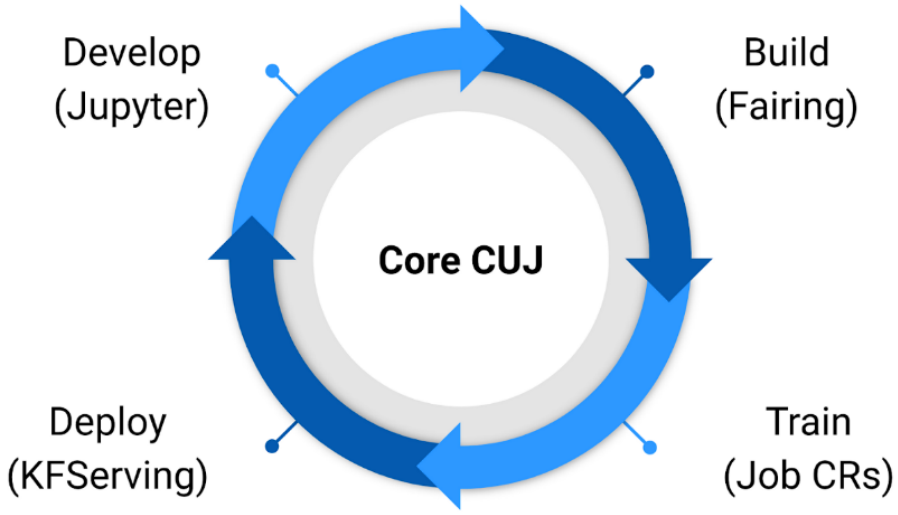
Figure 15-1 Phrases for training a model



Kubeflow was released in 2017, which is built on containers and Kubernetes. It aims to provide data scientists, machine learning engineers, and system O&M personnel with a platform for agile deployment, development, training, release, and management of machine learning services. It leverages the advantages of cloud native technologies to enable users to quickly and easily deploy, use, and manage the most popular machine learning software.

Kubeflow 1.0 is now available, providing capabilities in development, building, training, and deployment that cover the entire process of machine learning and deep learning for enterprise users.

The following shows an example.



With Kubeflow 1.0, you first develop a model using Jupyter, and then set up containers using tools such as Fairing (SDK). Next, you create Kubernetes resources to train the model. After the training is complete, you create and deploy servers for inference using KFServing. This is how you use Kubeflow to establish an end-to-end agile process of a machine learning task. This process can be fully automated using pipelines, which help achieve DevOps in the AI field.

Prerequisites

- A cluster named clusterA has been created on CCE. The cluster has an available GPU node that has two or more GPUs.
- EIPs have been bound to the nodes, and the kubectl command line tool has been configured. For details, see **Connecting to a Cluster Using kubectl**.

Installing Kustomize

Kustomize is an open-source tool used to manage the configuration of applications running in Kubernetes clusters. It allows you to modify application configuration. Starting with Kubeflow 1.3, all components should be deployed only using Kustomize.

Step 1 Install Kustomize. Kubeflow is incompatible with earlier versions of Kustomize. Therefore, only Kustomize 5 and later versions are supported. In this example, Kubeflow 5.1.0 is used.

```
curl -o install_kustomize.sh "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh"
sh install_kustomize.sh 5.1.0 .
```


The installation may take 3 to 5 minutes, and the information similar to the following will be displayed:

```
v5.1.0
kustomize installed to /root/kubeflow/./kustomize
```

Step 2 Move kustomize to the **/bin** directory so that the **kustomize** command can be used globally.

```
cp kustomize /bin/
```

----End

Installing Kubeflow

Perform the steps in this section to install all official Kubeflow components. After the installation, you can access the Kubeflow central dashboard. For details, see **Connecting to Kubeflow**.

Step 1 Install Kubeflow 1.7.0.

```
wget https://github.com/kubeflow/manifests/archive/refs/tags/v1.7.0.zip
unzip v1.7.0.zip
```

Step 2 Use Kustomize to create a YAML file for deploying Kubeflow.

```
cd ./manifests-1.7.0/
kustomize build example -o example.yaml
```

Step 3 Configure storage resources required by Kubeflow.

- katib-mysql
- mysql-pv-claim
- minio-pv-claim
- authservice-pvc

Some storage resources need to be configured during the installation. The storage configuration in the official example cannot take effect in CCE. This may result in the preceding PVC fail to be created. Therefore, create a PVC with the same name in the cluster in advance. In this example, the EVS disk is used. You can change the storage type as required.

Create the **pvc.yaml** file. The following is an example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: katib-mysql
  namespace: kubeflow
  annotations:
    everest.io/disk-volume-type: SAS    # EVS disk type
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # Region of the node where the application is to be deployed
    failure-domain.beta.kubernetes.io/zone: <your_zone>      # AZ of the node where the application is to be deployed
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
---
apiVersion: v1
kind: PersistentVolumeClaim
```



```
metadata:
  name: mysql-pv-claim
  namespace: kubeflow
  annotations:
    everest.io/disk-volume-type: SAS    # EVS disk type
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # Region of the node where the application is
to be deployed
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # AZ of the node where the application is to be
deployed
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: csi-disk
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: minio-pvc
  namespace: kubeflow
  annotations:
    everest.io/disk-volume-type: SAS    # EVS disk type
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # Region of the node where the application is
to be deployed
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # AZ of the node where the application is to be
deployed
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
  storageClassName: csi-disk
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: authservice-pvc
  namespace: istio-system
  annotations:
    everest.io/disk-volume-type: SAS    # EVS disk type
  labels:
    failure-domain.beta.kubernetes.io/region: <your_region>   # Region of the node where the application is
to be deployed
    failure-domain.beta.kubernetes.io/zone: <your_zone>       # AZ of the node where the application is to be
deployed
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk
```

Create a PVC.

```
kubectl apply -f pvc.yaml
```

Step 4 Create related resources.

```
kubectl apply -f example.yaml
```

NOTE

Official images may fail to be pulled due to network problems, and the ImagePullBackOff or FailedPullImage error may occur in the workload. In this case, add a proper image proxy.

Step 5 Check whether pods in all namespaces are running.

```
kubectl get pod -A
```

If an unexpected problem occurs during resource creation, rectify it by referring to **Common Issues**.

----End

Common Issues

- In some scenarios where CRD resources do not exist, the following information is displayed:

  ```
  error: resource mapping not found for name: "<RESOURCE_NAME>" namespace: "<SOME_NAMESPACE>" from "STDIN": no matches for kind "<CRD_NAME>" in version "<CRD_FULL_NAME>"
  ensure CRDs are installed first
  ```

  Solution:

  This is because kustomization creates CRs ahead of CRDs. If you encounter this error message, create the resource again.

- When a workload is created, an error message is displayed, indicating that there are too many pods on the node. The error message is displayed as follows:

  ```
  0/x nodes are available: x Too many pods.
  ```

  Solution:

  This message indicates that the number of schedulable pods on the node exceeds the node's upper limit. To solve this problem, increase the number of nodes.

- The **training-operator** workload cannot run properly. The error message in the log is displayed as follows:

  ```
  Waited for 1.039518449s due to client-side throttling, not priority and fairness, request: GET:https://10.247.0.1:443/apis/xxx/xx?timeout=32s
  ```

  Solution:

  Run the following command to check the statuses of the unavailable APIServices in the cluster:

  ```
  kubectl get apiservice
  ```

  If there is no APIService in the **FALSE** state, the **training-operator** workload will run 1 to 2 minutes later.

15.1.2 Training a TensorFlow Model

After Kubeflow is deployed, it is easy to use the ps-worker mode to train TensorFlow models. This section describes an official TensorFlow training example provided by Kubeflow. For details, see **TensorFlow Training (TFJob)**.

Running the Mnist Example

Step 1 Deploy the TFJob resource to start training.

Create the **tf-mnist.yaml** file. The following is an example:

```
apiVersion: "kubeflow.org/v1"
kind: TFJob
metadata:
  name: tfjob-simple
  namespace: kubeflow
spec:
```



```
tfReplicaSpecs:
  Worker:
    replicas: 2
    restartPolicy: OnFailure
    template:
      spec:
        containers:
          - name: tensorflow
            image: kubeflow/tf-mnist-with-summaries:latest
            command:
              - "python"
              - "/var/tf_mnist/mnist_with_summaries.py"
```

Step 2 Create the TFJob.

```
kubectl apply -f tf-mnist.yaml
```

Step 3 View the logs after the worker running is complete.

```
kubectl -n kubeflow logs tfjob-simple-worker-0
```

Information similar to the following is displayed:

```
...
Accuracy at step 900: 0.964
Accuracy at step 910: 0.9653
Accuracy at step 920: 0.9665
Accuracy at step 930: 0.9681
Accuracy at step 940: 0.9664
Accuracy at step 950: 0.9667
Accuracy at step 960: 0.9694
Accuracy at step 970: 0.9683
Accuracy at step 980: 0.9687
Accuracy at step 990: 0.966
Adding run metadata for 999
```

Step 4 Delete the TFJob.

```
kubectl delete -f tf-mnist.yaml
```

----End

Using a GPU

The training can be performed in the GPU scenario. In this scenario, the cluster must contain GPU nodes and proper drivers must be installed.

Step 1 Specify the GPU resources in the TFJob.

Create the **tf-gpu.yaml** file. The following is an example:

This example runs in the TensorFlow distributed architecture. The ResNet50 model in the convolutional neural network (CNN) is used to train randomly generated images. A total of **32** (**batch_size**) images are trained each time, and the images are trained **100** times in total. Additionally, the performance (**image/sec**) of each training is recorded.

```
apiVersion: "kubeflow.org/v1"
kind: "TFJob"
metadata:
  name: "tf-smoke-gpu"
spec:
  tfReplicaSpecs:
    PS:
      replicas: 1
      template:
        metadata:
          creationTimestamp: null
```



```
    spec:
      containers:
        - args:
            - python
            - tf_cnn_benchmarks.py
            - --batch_size=32
            - --model=resnet50
            - --variable_update=parameter_server
            - --flush_stdout=true
            - --num_gpus=1
            - --local_parameter_device=cpu
            - --device=cpu
            - --data_format=NHWC
          image: docker.io/kubeflow/tf-benchmarks-cpu:v20171202-bdab599-dirty-284af3
          name: tensorflow
          ports:
            - containerPort: 2222
              name: tfjob-port
          resources:
            limits:
              cpu: "1"
          workingDir: /opt/tf-benchmarks/scripts/tf_cnn_benchmarks
      restartPolicy: OnFailure
  Worker:
    replicas: 1
    template:
      metadata:
        creationTimestamp: null
      spec:
        containers:
          - args:
              - python
              - tf_cnn_benchmarks.py
              - --batch_size=32
              - --model=resnet50
              - --variable_update=parameter_server
              - --flush_stdout=true
              - --num_gpus=1
              - --local_parameter_device=cpu
              - --device=gpu
              - --data_format=NHWC
            image: docker.io/kubeflow/tf-benchmarks-gpu:v20171202-bdab599-dirty-284af3
            name: tensorflow
            ports:
              - containerPort: 2222
                name: tfjob-port
            resources:
              limits:
                nvidia.com/gpu: 1       # Number of GPUs
            workingDir: /opt/tf-benchmarks/scripts/tf_cnn_benchmarks
        restartPolicy: OnFailure
```

Step 2 Create the TFJob.

```
kubectl apply -f tf-gpu.yaml
```

Step 3 After the worker runs the job (about 5 minutes if a GPU is used), run the following command to view the result.

```
kubectl logs tf-smoke-gpu-worker-0
```

Information similar to the following is displayed:

```
...
INFO|2023-09-02T12:04:25|/opt/launcher.py|27| Running warm up
INFO|2023-09-02T12:08:55|/opt/launcher.py|27| Done warm up
INFO|2023-09-02T12:08:55|/opt/launcher.py|27| Step      Img/sec loss
INFO|2023-09-02T12:08:56|/opt/launcher.py|27| 1 images/sec: 68.8 +/- 0.0 (jitter = 0.0) 8.777
INFO|2023-09-02T12:09:00|/opt/launcher.py|27| 10        images/sec: 70.4 +/- 0.4 (jitter = 1.8) 8.557
INFO|2023-09-02T12:09:04|/opt/launcher.py|27| 20        images/sec: 70.5 +/- 0.3 (jitter = 1.5) 8.090
INFO|2023-09-02T12:09:09|/opt/launcher.py|27| 30        images/sec: 70.3 +/- 0.3 (jitter = 1.6) 8.041
```



```
INFO|2023-09-02T12:09:13|/opt/launcher.py|27| 40        images/sec: 70.1 +/- 0.2 (jitter = 1.7) 9.464
INFO|2023-09-02T12:09:18|/opt/launcher.py|27| 50        images/sec: 70.1 +/- 0.2 (jitter = 1.6) 7.797
INFO|2023-09-02T12:09:23|/opt/launcher.py|27| 60        images/sec: 70.1 +/- 0.2 (jitter = 1.6) 8.595
INFO|2023-09-02T12:09:27|/opt/launcher.py|27| 70        images/sec: 70.0 +/- 0.2 (jitter = 1.7) 7.853
INFO|2023-09-02T12:09:32|/opt/launcher.py|27| 80        images/sec: 69.9 +/- 0.2 (jitter = 1.7) 7.849
INFO|2023-09-02T12:09:36|/opt/launcher.py|27| 90        images/sec: 69.8 +/- 0.2 (jitter = 1.7) 7.911
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| 100       images/sec: 69.7 +/- 0.1 (jitter = 1.7) 7.853
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| ----------------------------------------------------------------
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| total images/sec: 69.68
INFO|2023-09-02T12:09:41|/opt/launcher.py|27| ----------------------------------------------------------------
INFO|2023-09-02T12:09:42|/opt/launcher.py|80| Finished: python tf_cnn_benchmarks.py --batch_size=32 --
model=resnet50 --variable_update=parameter_server --flush_stdout=true --num_gpus=1 --
local_parameter_device=cpu --device=gpu --data_format=NHWC --job_name=worker --ps_hosts=tf-smoke-
gpu-ps-0.default.svc:2222 --worker_hosts=tf-smoke-gpu-worker-0.default.svc:2222 --task_index=0
INFO|2023-09-02T12:09:42|/opt/launcher.py|84| Command ran successfully sleep for ever.
```

The training performance of a single GPU is **69.68** images per second.

----End

15.1.3 Using Kubeflow and Volcano to Train an AI Model

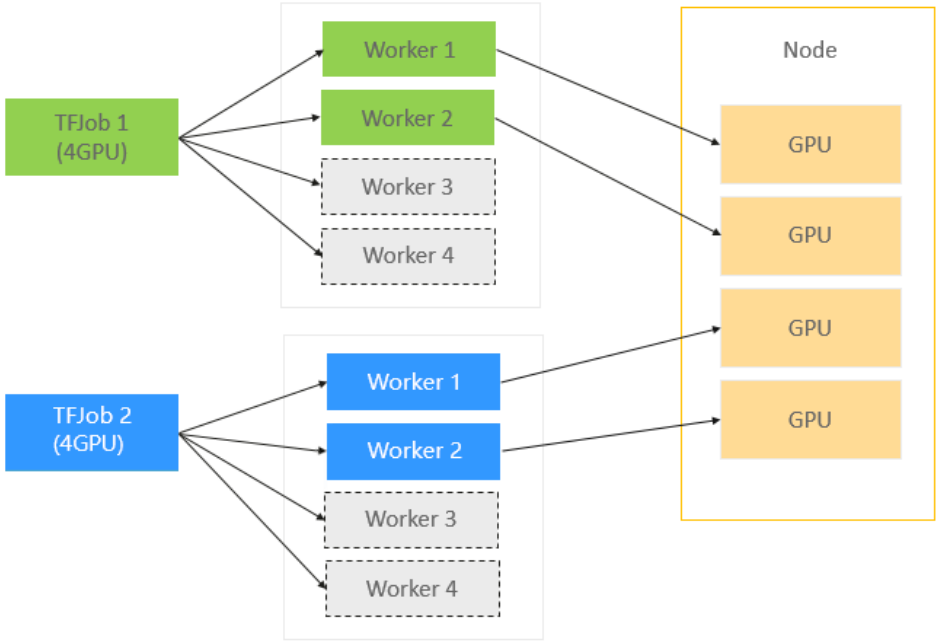
Kubernetes has become the de facto standard for cloud native application orchestration and management. An increasing number of applications are migrated to Kubernetes. AI and machine learning inherently involve a large number of computing-intensive tasks. Kubernetes is a preferential tool for developers building AI platforms because of its excellent capabilities in resource management, application orchestration, and O&M monitoring.

Kubernetes Pain Points

Kubeflow uses the default scheduler of Kubernetes, which was initially designed for long-term running services. Its scheduling capability is inadequate for tasks that involve batch computing and elastic scheduling in AI and big data scenarios. The main constraints are as follows:

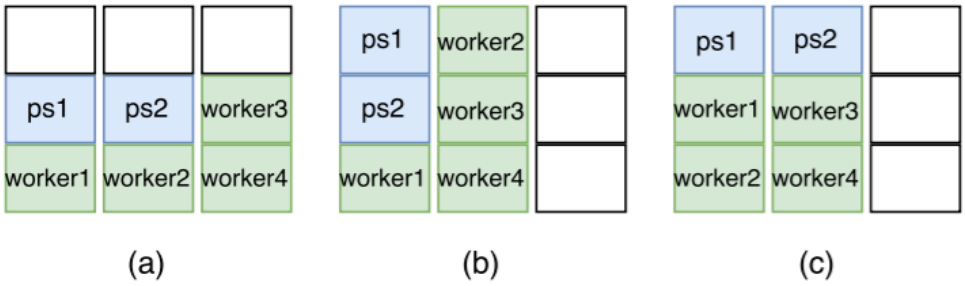
Resource preemption

A TensorFlow job involves two roles: parameter server (ps) and worker. Only when pods of these two roles run properly at the same time can a TensorFlow job be executed normally. However, the default scheduler is insensitive to the roles of pods in a TensorFlow job. Pods are treated identically and scheduled one by one. This causes problems when there are multiple jobs to schedule and cluster resources are scarce. Each job could end up being allocated with only part of the resources it needs to finish the execution. That is, resources are used up while no job can be successfully executed. To better illustrate this dilemma, assume that you want to run two TensorFlow jobs, namely, TFJob1 and TFJob2. Each of these jobs has four workers, which means each job requires four GPUs to run. However, your cluster only has four available GPUs in total. In this case, with the default scheduler, TFJob1 and TFJob2 could end up being allocated two GPUs each. They are waiting each other to finish and release the resources. However, this will not happen until you manually intervene. The deadlock created in this situation cause resource wastes and low efficiency in job execution.



Lack of affinity-based scheduling

In distributed training, data is frequently exchanged between parameter servers and workers. To ensure higher efficiency, parameter servers and workers of the same job should be scheduled to the same node for faster transmission using local networks. However, the default scheduler is insensitive to the affinity between parameter servers and workers of the same job. Pods are randomly scheduled instead. As shown in the following figure, assume that you want to run two TensorFlow jobs with each having one ps and two workers. With the default scheduler, the scheduling results could be any of the following three situations. However, only result (c) can deliver the highest efficiency. In (c), the ps and the workers can use the local network to communicate more efficiently and shorten the training time.



Volcano, a Perfect Batch Scheduling System for Accelerating AI Computing

Volcano is an enhanced batch scheduling system for high-performance computing workloads running on Kubernetes. It complements Kubernetes in machine learning, deep learning, HPC, and big data computing scenarios, providing capabilities such as gang scheduling, computing task queue management, task-topology, and GPU affinity scheduling. In addition, Volcano enhances batch task

creation and lifecycle management, fair-share, binpack, and other Kubernetes-native capabilities. It fully addresses the constraints of Kubeflow in distributed training mentioned above.

For more information about Volcano, visit **https://github.com/volcano-sh/volcano**.

Using Volcano in Huawei Cloud

The convergence of Kubeflow and Volcano, two open-source projects, greatly simplifies and accelerates AI computing workloads running on Kubernetes. The two projects have been recognized by an increasing number of players in the field and applied in production environments. Volcano is used in Huawei Cloud CCE, Cloud Container Instance (CCI), and Kubernetes-Native Batch Computing Solution. Volcano will continue to iterate with optimized algorithms, enhanced capabilities such as intelligent scheduling, and new inference features such as GPU Share, to further improve the efficiency of Kubeflow batch training and inference.

Implementing Typical Distributed AI Training Jobs

This section describes how to perform distributed training of a digital image classification model using the MNIST dataset based on Kubeflow and Volcano.

Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 Deploy volcano on the cluster.

In the navigation pane, choose **Add-ons**. Click **Install** under the **volcano** add-on. In the window that slides out from the right, configure the specifications and click **Install**.

Step 3 Deploy the MNIST dataset.

1. Download **kubeflow/examples** to the local host and select an operation guide based on the environment.
   ```
   yum install git
   git clone https://github.com/kubeflow/examples.git
   ```
2. Install python3.
   ```
   wget https://www.python.org/ftp/python/3.6.8/Python-3.6.8.tgz
   tar -zxvf Python-3.6.8.tgz
   ```



```
cd Python-3.6.8 ./configure
make make install
```

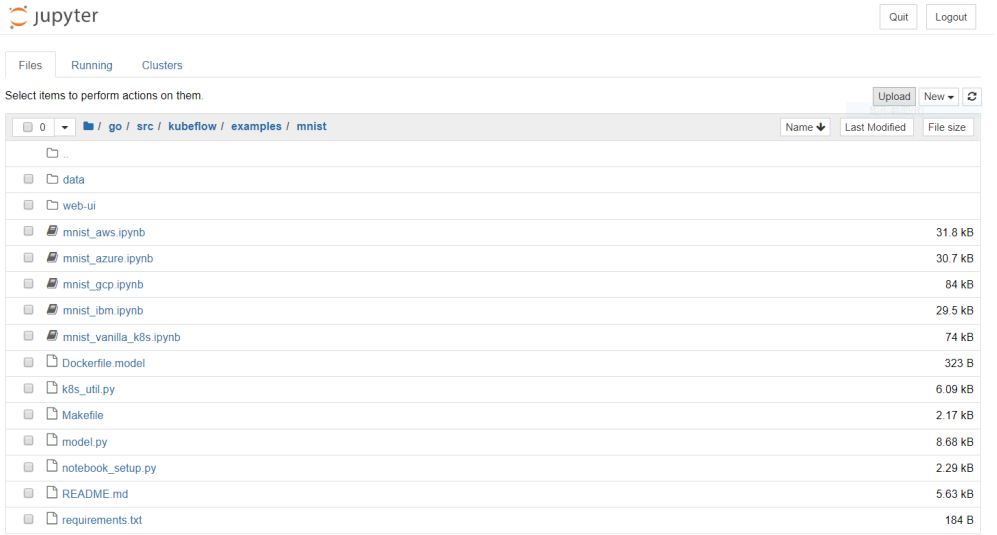
After the installation, run the following commands to check whether the installation is successful:

```
python3 -V
pip3 -V
```

3. Install and start Jupyter Notebook.

```
pip3 install jupyter notebook
jupyter notebook --allow-root
```

4. Configure an SSH tunnel on PuTTY and remotely connect to the notebook.
5. After the connection is successful, enter **localhost:8000** in the address box of a browser to log in to the notebook.



6. Create a distributed training job as prompted by Jupyter. Set the value of **schedulerName** to **volcano** to enable volcano.

```
kind: TFJob
metadata:
  name: {train_name}
spec:
  schedulerName: volcano
  tfReplicaSpecs:
    Ps:
      replicas: {num_ps}
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          serviceAccount: default-editor
          containers:
          - name: tensorflow
            command:
              ...
            env:
              ...
            image: {image}
            workingDir: /opt
          restartPolicy: OnFailure
    Worker:
      replicas: 1
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
```



```
  spec:
    serviceAccount: default-editor
    containers:
    - name: tensorflow
      command:
        ...
      env:
        ...
      image: {image}
      workingDir: /opt
    restartPolicy: OnFailure
```

Step 4 Submit the job and start the training.

```
kubectl apply -f mnist.yaml
```

```
root@ecs-1953:~/tmp# kubectl get po
NAME                           READY   STATUS    RESTARTS   AGE
tensorflow-mnist-ps-0          1/1     Running   0          5s
tensorflow-mnist-worker-0      1/1     Running   0          5s
```

After the training job is complete, you can query the training results on the Kubeflow UI. This is how you run a simple distributed training job using Kubeflow and Volcano. Kubeflow simplifies TensorFlow job configuration. Volcano, with simply one more line of configuration, saves you significant time and effort in large-scale distributed training by providing capabilities such as gang scheduling and task topology to eliminate deadlocks and achieve affinity scheduling.

----End

15.2 Running Caffe in CCE

15.2.1 Prerequisites

This section provides an example for running Caffe on CCE to classify an image. For more information, see **https://github.com/BVLC/caffe/blob/master/examples/00-classification.ipynb**.

Pre-configuring OBS Storage Data

Create an OBS bucket and ensure that the following folders have been created and required files have been uploaded to the specified paths using the OBS Browser.

The folder name can be in the format of *File path in the bucket/File name*. You can search for the file download addresses in the specified paths of the specified project in GitHub, as shown in **1** and **2**.

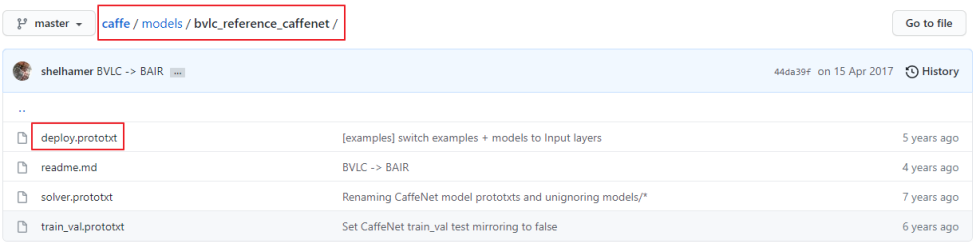
1. models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel

   https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet

name	caffemodel	caffemodel_url	license
BAIR/BVLC CaffeNet Model	bvlc_reference_caffenet.caffemodel	http://dl.caffe.berkeleyvision.org/bvlc_reference_caffenet.caffemodel	unrestricted

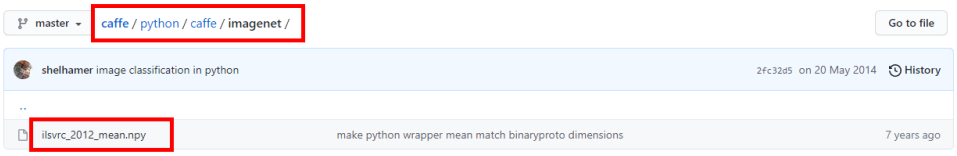
2. models/bvlc_reference_caffenet/deploy.prototxt

   https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet



3. python/caffe/imagenet/ilsvrc_2012_mean.npy

   https://github.com/BVLC/caffe/tree/master/python/caffe/imagenet



4. outputimg/

   An empty folder **outputimg** is created to store output files.

5. examples/images/cat.jpg

   https://github.com/BVLC/caffe/blob/master/examples/00-classification.ipynb

   Save the picture of the cat in the link.

6. data/ilsvrc12/*

   https://github.com/BVLC/caffe/tree/master/data/ilsvrc12

   Obtain and execute the **get_ilsvrc_aux.sh** script. The script downloads a compressed package and decompresses it. After the script is executed, upload all decompressed files to the directory.

7. caffeEx00.py

```
# set up Python environment: numpy for numerical routines, and matplotlib for plotting
import numpy as np
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
# display plots in this notebook
#%matplotlib inline

# set display defaults
plt.rcParams['figure.figsize'] = (10, 10)        # large images
plt.rcParams['image.interpolation'] = 'nearest'  # don't interpolate: show square pixels
plt.rcParams['image.cmap'] = 'gray'  # use grayscale output rather than a (potentially misleading)
color heatmap

# The caffe module needs to be on the Python path;
#  we'll add it here explicitly.
import sys
caffe_root = '/home/'  # this file should be run from {caffe_root}/examples (otherwise change this
line)
sys.path.insert(0, caffe_root + 'python')

import caffe
# If you get "No module named _caffe", either you have not built pycaffe or you have the wrong path.

import os
```



```
#if os.path.isfile(caffe_root + 'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel'):
#    print 'CaffeNet found.'
#else:
#    print 'Downloading pre-trained CaffeNet model...'
#    !../scripts/download_model_binary.py ../models/bvlc_reference_caffenet

caffe.set_mode_cpu()

model_def = caffe_root + 'models/bvlc_reference_caffenet/deploy.prototxt'
model_weights = caffe_root + 'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel'

net = caffe.Net(model_def,      # defines the structure of the model
                model_weights,  # contains the trained weights
                caffe.TEST)     # use test mode (e.g., don't perform dropout)

# load the mean ImageNet image (as distributed with Caffe) for subtraction
mu = np.load(caffe_root + 'python/caffe/imagenet/ilsvrc_2012_mean.npy')
mu = mu.mean(1).mean(1)  # average over pixels to obtain the mean (BGR) pixel values
print 'mean-subtracted values:', zip('BGR', mu)

# create transformer for the input called 'data'
transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})

transformer.set_transpose('data', (2,0,1))  # move image channels to outermost dimension
transformer.set_mean('data', mu)            # subtract the dataset-mean value in each channel
transformer.set_raw_scale('data', 255)      # rescale from [0, 1] to [0, 255]
transformer.set_channel_swap('data', (2,1,0))  # swap channels from RGB to BGR

# set the size of the input (we can skip this if we're happy
#  with the default; we can also change it later, e.g., for different batch sizes)
net.blobs['data'].reshape(50,        # batch size
                          3,         # 3-channel (BGR) images
                          227, 227)  # image size is 227x227

image = caffe.io.load_image(caffe_root + 'examples/images/cat.jpg')
transformed_image = transformer.preprocess('data', image)
plt.imshow(image)
plt.savefig(caffe_root + 'outputimg/img1.png')

# copy the image data into the memory allocated for the net
net.blobs['data'].data[...] = transformed_image

### perform classification
output = net.forward()

output_prob = output['prob'][0]  # the output probability vector for the first image in the batch

print 'predicted class is:', output_prob.argmax()

# load ImageNet labels
labels_file = caffe_root + 'data/ilsvrc12/synset_words.txt'
#if not os.path.exists(labels_file):
#    !../data/ilsvrc12/get_ilsvrc_aux.sh

labels = np.loadtxt(labels_file, str, delimiter='\t')

print 'output label:', labels[output_prob.argmax()]

# sort top five predictions from softmax output
top_inds = output_prob.argsort()[::-1][:5]  # reverse sort and take five largest items

print 'probabilities and labels:'
zip(output_prob[top_inds], labels[top_inds])
```


15.2.2 Preparing Resources

Adding a GPU Node to a Cluster

Step 1 Log in to the CCE console and click the name of the target cluster to access the cluster console.

Step 2 Install the GPU add-on.

1. In the navigation pane, choose **Add-ons**. Locate **gpu-beta** (or **gpu-device-plugin**) and click **Install**.
2. In the window that slides out from the right, configure the key parameters.
   - **NVIDIA Driver**: Enter the download link of the NVIDIA driver. Select a driver based on the graphics card model of the GPU node.

   Retain the default values for other parameters. For details, see **gpu-beta**.
3. Click **Install**.

Step 3 Create a GPU node.

1. In the navigation pane, choose **Nodes**. Click **Create Node** in the upper right corner and configure the parameters.
2. Select **GPU-accelerated** for node specifications and configure other parameters as required. For details, see **Creating a Node**.
3. After the configuration, click **Next: Confirm**. On the page displayed, confirm the configuration and click **Submit**.

Step 4 View its status in the node list.

----End

Importing an OBS Volume

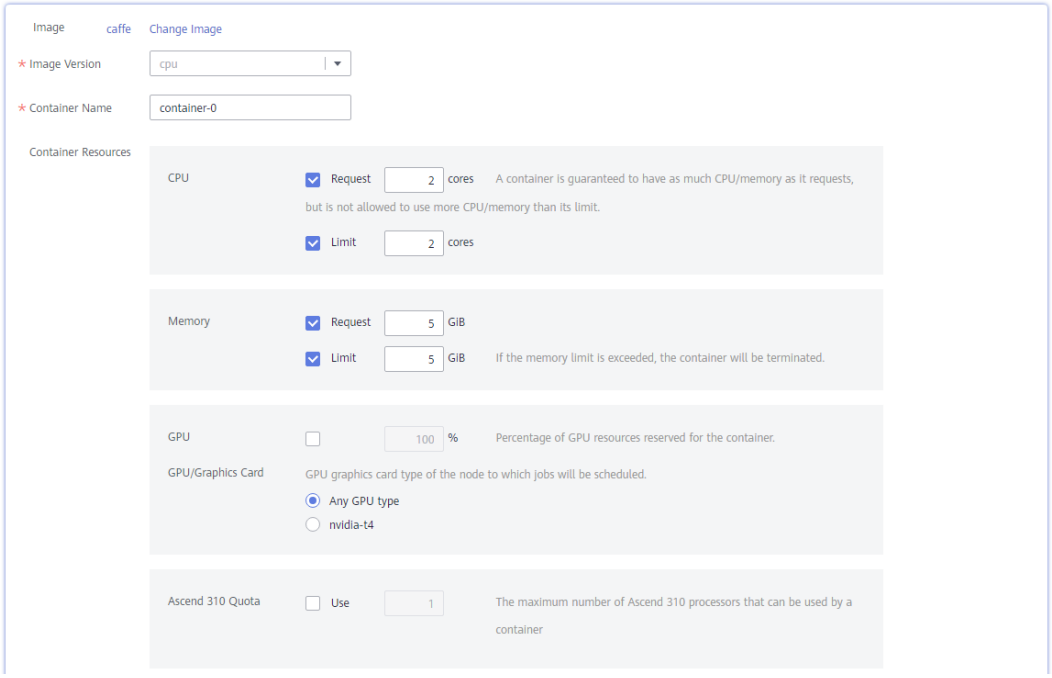
Go to the storage management page and import the OBS volume created in **Pre-configuring OBS Storage Data**.

15.2.3 Caffe Classification Example

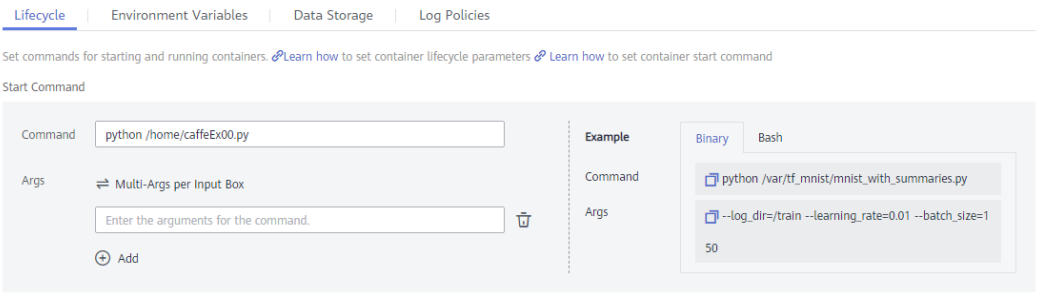
This section uses the official Caffe classification example at **https://github.com/BVLC/caffe/blob/master/examples/00-classification.ipynb** to illustrate how to run Caffe jobs on CCE.

Using CPUs

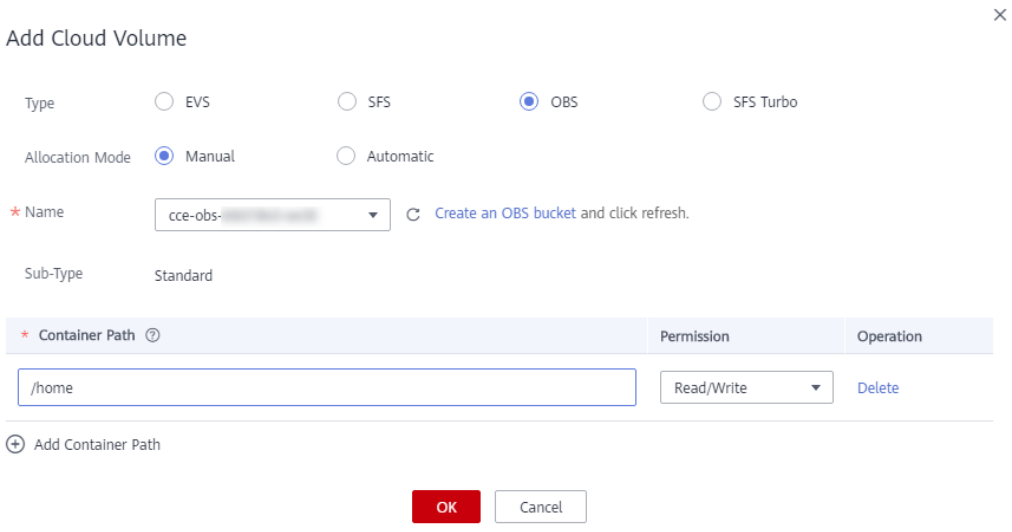
Create a job using the third-party **image bvlc/caffe:cpu**. Set the container specifications.



Add the startup command **python /home/caffeEx00.py**.

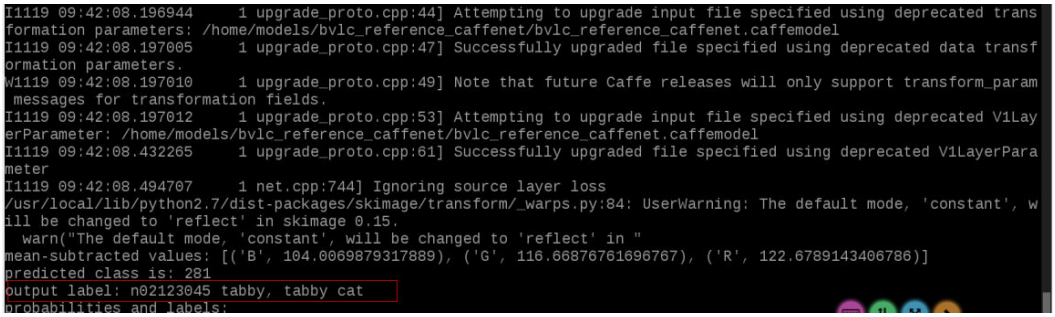


Mount the imported OBS volume.



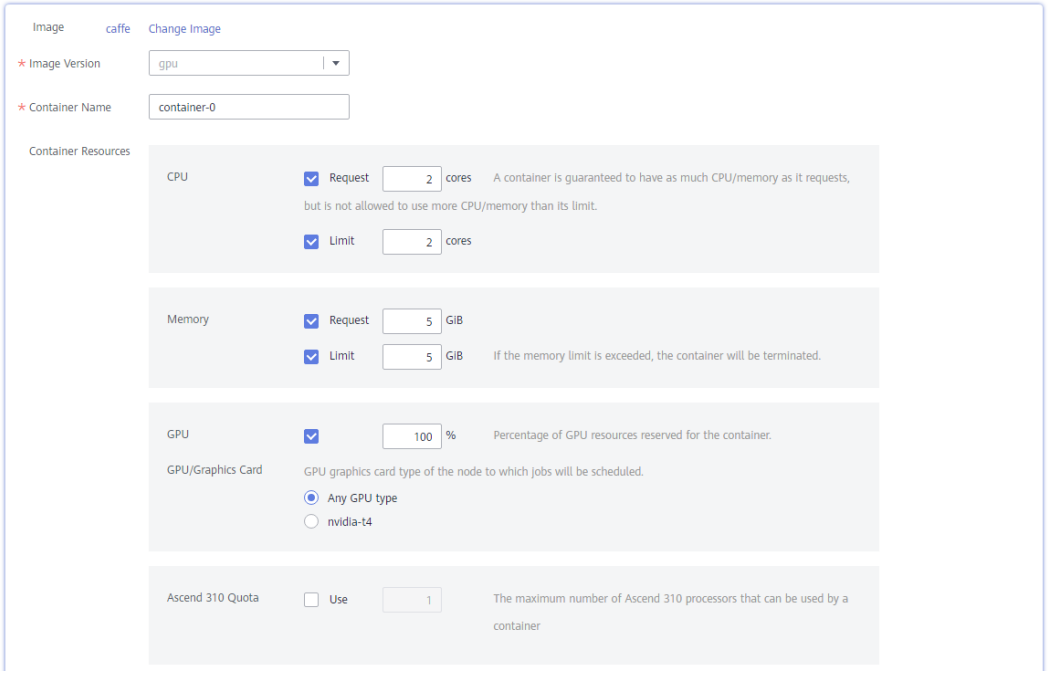
Click **Create**. After the job execution is complete, go to the **outputimg** directory of the OBS volume to view the image used for inference.

Log in to the node added in **Adding a GPU Node to a Cluster** and run the **docker logs {*Container ID*}** command to view the classification result. The result is displayed as **tabby cat**.

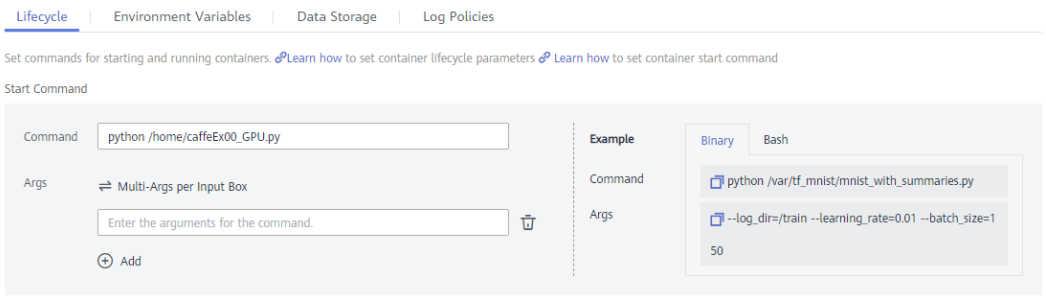


Using GPUs

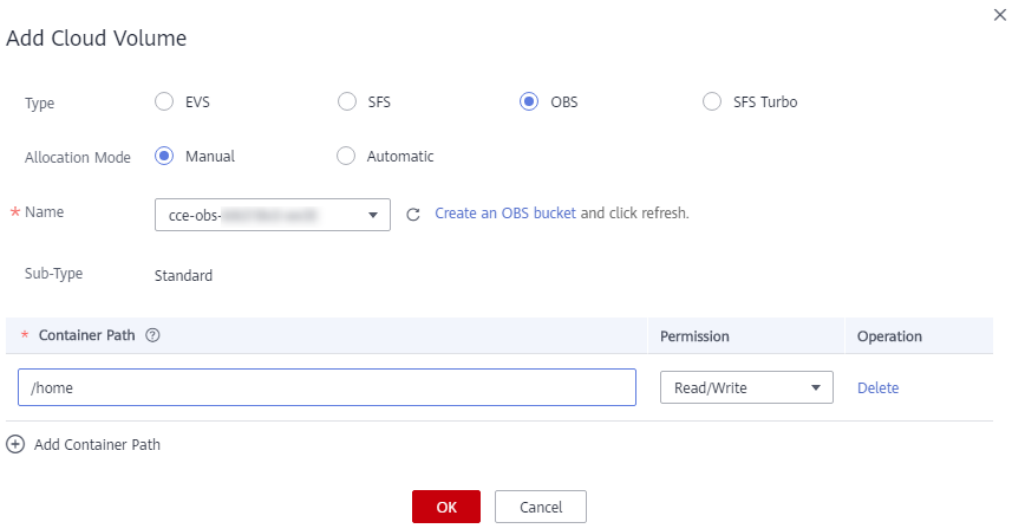
Create a job using the third-party **bvlc/caffe:gpu**. Set the container specifications.



Add the startup command **python /home/caffeEx00_GPU.py**.

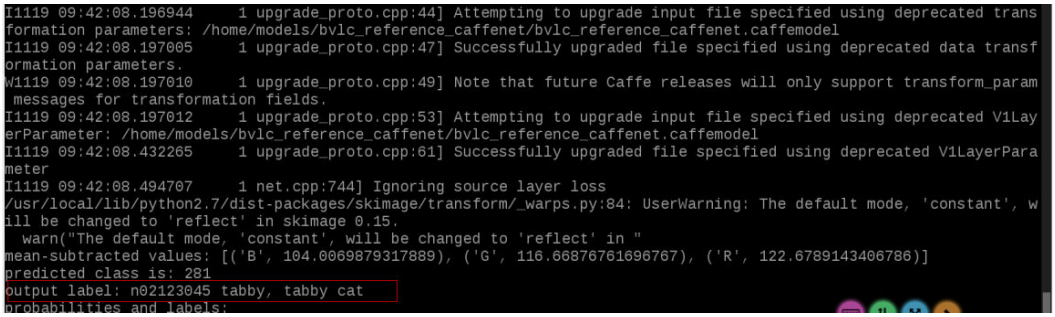


Mount the imported OBS volume.



Click **Create**. After the job execution is complete, go to the **outputimg** directory of the OBS volume to view the image used for inference.

Log in to the node added in **Adding a GPU Node to a Cluster** and run the **docker logs {***Container ID***}** command to view the classification result. The result is displayed as **tabby cat**.



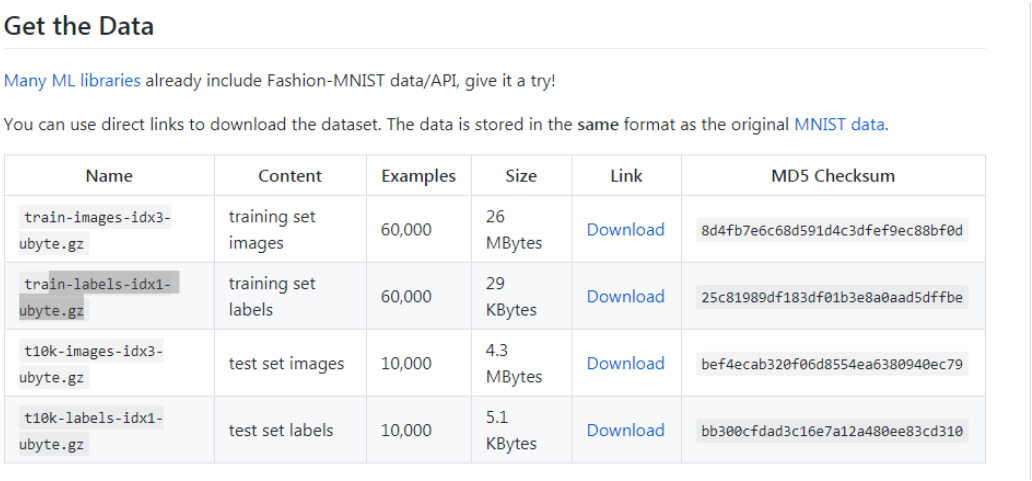
15.3 Running TensorFlow in CCE

Preparing Resources

- Create a CCE cluster and GPU nodes, and use the gpu-beta add-on to install the graphics card driver.
- Add an object storage volume to the cluster.

Pre-configuring Data

Download data from **https://github.com/zalandoresearch/fashion-mnist**.



Get the Data

Many ML libraries already include Fashion-MNIST data/API, give it a try!

You can use direct links to download the dataset. The data is stored in the **same** format as the original MNIST data.

Name	Content	Examples	Size	Link	MD5 Checksum
train-images-idx3-ubyte.gz	training set images	60,000	26 MBytes	Download	8d4fb7e6c68d591d4c3dfef9ec88bf0d
train-labels-idx1-ubyte.gz	training set labels	60,000	29 KBytes	Download	25c81989df183df01b3e8a0aad5dffbe
t10k-images-idx3-ubyte.gz	test set images	10,000	4.3 MBytes	Download	bef4ecab320f06d8554ea6380940ec79
t10k-labels-idx1-ubyte.gz	test set labels	10,000	5.1 KBytes	Download	bb300cfdad3c16e7a12a480ee83cd310

Obtain the TensorFlow machine learning (ML) example and modify it based on your requirements.

basicClass.py

```
# TensorFlow and tf.keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import gzip
from tensorflow.python.keras.utils import get_file
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt

print(tf.__version__)

#fashion_mnist = keras.datasets.fashion_mnist
#(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()

def load_data():
    base = "file:////home/data/"
    files = [
        'train-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz',
        't10k-labels-idx1-ubyte.gz', 't10k-images-idx3-ubyte.gz'
    ]

    paths = []
    for fname in files:
        paths.append(get_file(fname, origin=base + fname))

    with gzip.open(paths[0], 'rb') as lbpath:
        y_train = np.frombuffer(lbpath.read(), np.uint8, offset=8)

    with gzip.open(paths[1], 'rb') as imgpath:
        x_train = np.frombuffer(
            imgpath.read(), np.uint8, offset=16).reshape(len(y_train), 28, 28)

    with gzip.open(paths[2], 'rb') as lbpath:
        y_test = np.frombuffer(lbpath.read(), np.uint8, offset=8)

    with gzip.open(paths[3], 'rb') as imgpath:
        x_test = np.frombuffer(
            imgpath.read(), np.uint8, offset=16).reshape(len(y_test), 28, 28)

    return (x_train, y_train), (x_test, y_test)

(train_images, train_labels), (test_images, test_labels) = load_data()
```

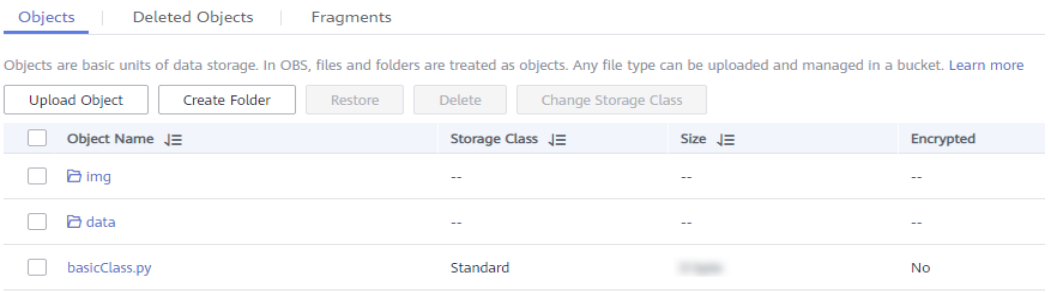


```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.savefig('/home/img/basicimg1.png')

train_images = train_images / 255.0

test_images = test_images / 255.0

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.savefig('/home/img/basicimg2.png')

model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)

predictions = model.predict(test_images)

def plot_image(i, predictions_array, true_label, img):
  predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.binary)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array[i], true_label[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)
```



```
  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')

i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.savefig('/home/img/basicimg3.png')

i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions,  test_labels)
plt.savefig('/home/img/basicimg4.png')

# Plot the first X test images, their predicted label, and the true label
# Color correct predictions in blue, incorrect predictions in red
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions, test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions, test_labels)
plt.savefig('/home/img/basicimg5.png')
```

Go to the OBS bucket page, create the **data** and **img** folders, and upload **basicClass.py**.

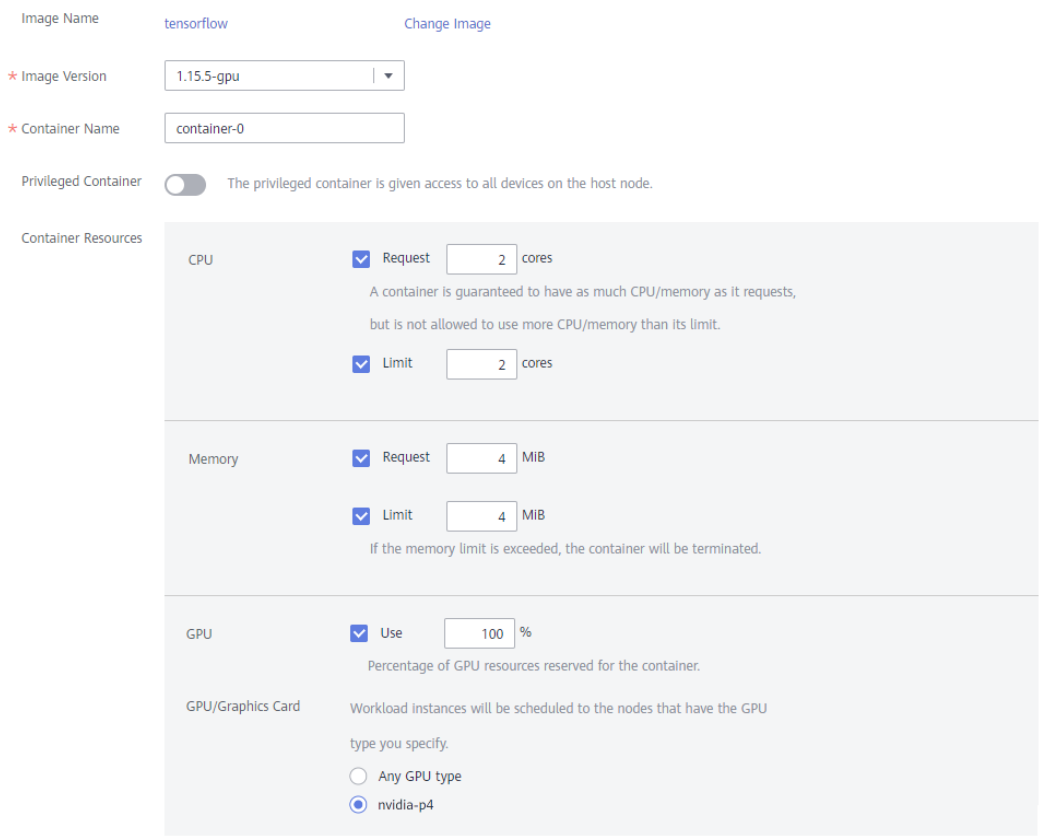


Go to the **data** folder and upload the four .gz files downloaded from GitHub.

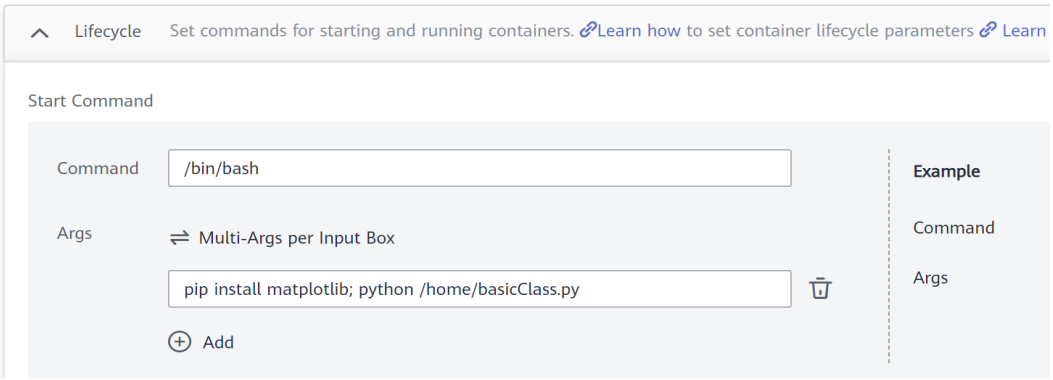
ML Example

In this section, the ML example from the TensorFlow official website is used. For details, see **https://www.tensorflow.org/tutorials/keras/classification?hl=en-us**.

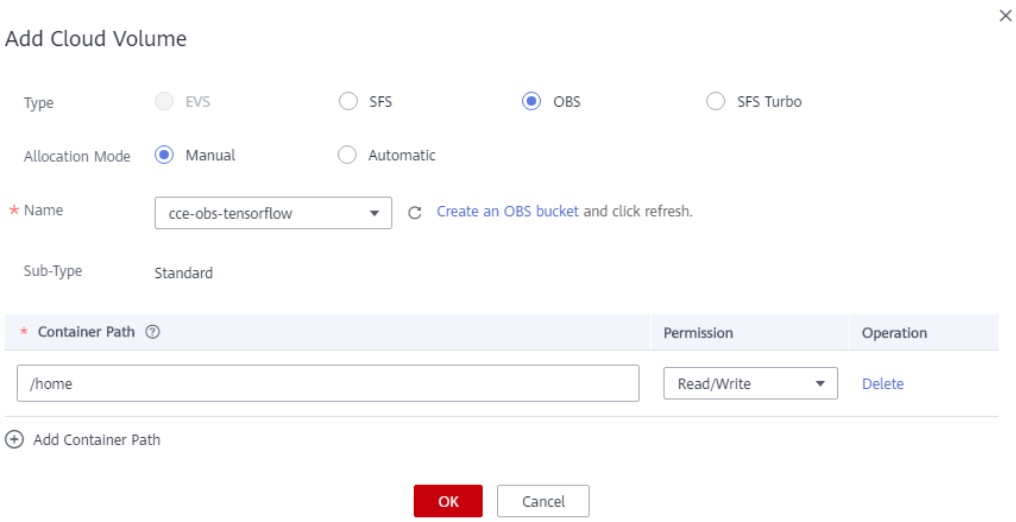
Create a job using the third-party **tensorflow/tensorflow:1.15.5-gpu**. Set the container specifications.



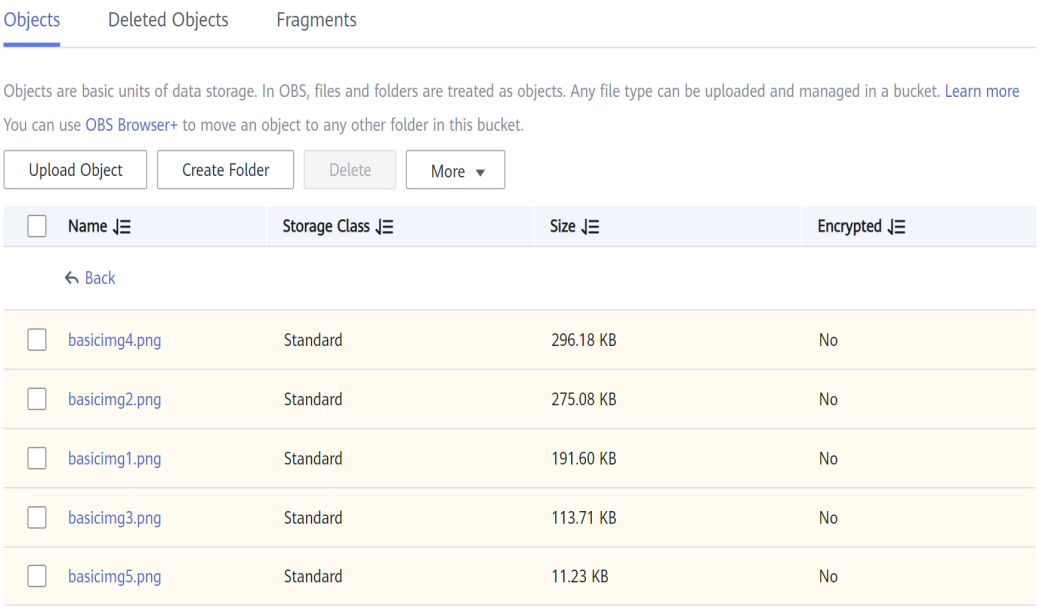
Add **pip install matplotlib;python /home/basicClass.py** in the **Start Command** area.



Mount the created OBS volume.



Click **Create**. Wait until the job execution is complete. On the OBS page, you can view the execution results that are shown as images.



If you want to use kubectl, you can use the following example YAML:

```
kind: Job
apiVersion: batch/v1
metadata:
  name: testjob
  namespace: default
spec:
  parallelism: 1
  completions: 1
  backoffLimit: 6
  template:
    metadata:
      name: testjob
    spec:
      volumes:
        - name: cce-obs-tensorflow
          persistentVolumeClaim:
```



```
        claimName: cce-obs-tensorflow
    containers:
      - name: container-0
        image: 'tensorflow/tensorflow:1.15.5-gpu'
        restartPolicy: OnFailure
        command:
          - /bin/bash
        args:
          - '-c'
          - pip install matplotlib;python /home/basicClass.py
        resources:
          limits:
            cpu: '2'
            memory: 4Gi
            nvidia.com/gpu: '1'
          requests:
            cpu: '2'
            memory: 4Gi
            nvidia.com/gpu: '1'
        volumeMounts:
          - name: cce-obs-tensorflow
            mountPath: /home
        imagePullPolicy: IfNotPresent
    imagePullSecrets:
      - name: default-secret
```

15.4 Running Flink in CCE

This section describes how to deploy a Flink cluster on CCE and run WordCount jobs.

Prerequisites

A CCE cluster with available nodes has been created. An elastic IP address has been bound to the nodes, and kubectl has been configured.

Deployment Process

For details about the process, see **https://ci.apache.org/projects/flink/flink-docs-stable/ops/deployment/kubernetes.html**.

Creating a Flink Session Cluster

Create two Deployments, one Service, and one ConfigMap by following the instructions provided on the preceding web page.

flink-configuration-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: flink-config
  labels:
    app: flink
data:
  flink-conf.yaml: |+
    jobmanager.rpc.address: flink-jobmanager
    taskmanager.numberOfTaskSlots: 2
    blob.server.port: 6124
    jobmanager.rpc.port: 6123
    taskmanager.rpc.port: 6122
    queryable-state.proxy.ports: 6125
```



```
    jobmanager.memory.process.size: 1600m
    taskmanager.memory.process.size: 1728m
    parallelism.default: 2
  log4j-console.properties: |+
    # This affects logging for both user code and Flink
    rootLogger.level = INFO
    rootLogger.appenderRef.console.ref = ConsoleAppender
    rootLogger.appenderRef.rolling.ref = RollingFileAppender

    # Uncomment this if you want to _only_ change Flink's logging
    #logger.flink.name = org.apache.flink
    #logger.flink.level = INFO

    # The following lines keep the log level of common libraries/connectors on
    # log level INFO. The root logger does not override this. You have to manually
    # change the log levels here.
    logger.akka.name = akka
    logger.akka.level = INFO
    logger.kafka.name= org.apache.kafka
    logger.kafka.level = INFO
    logger.hadoop.name = org.apache.hadoop
    logger.hadoop.level = INFO
    logger.zookeeper.name = org.apache.zookeeper
    logger.zookeeper.level = INFO

    # Log all infos to the console
    appender.console.name = ConsoleAppender
    appender.console.type = CONSOLE
    appender.console.layout.type = PatternLayout
    appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x - %m%n

    # Log all infos in the given rolling file
    appender.rolling.name = RollingFileAppender
    appender.rolling.type = RollingFile
    appender.rolling.append = false
    appender.rolling.fileName = ${sys:log.file}
    appender.rolling.filePattern = ${sys:log.file}.%i
    appender.rolling.layout.type = PatternLayout
    appender.rolling.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x - %m%n
    appender.rolling.policies.type = Policies
    appender.rolling.policies.size.type = SizeBasedTriggeringPolicy
    appender.rolling.policies.size.size=100MB
    appender.rolling.strategy.type = DefaultRolloverStrategy
    appender.rolling.strategy.max = 10

    # Suppress the irrelevant (wrong) warnings from the Netty channel handler
    logger.netty.name = org.apache.flink.shaded.akka.org.jboss.netty.channel.DefaultChannelPipeline
    logger.netty.level = OFF
```

jobmanager-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: flink-jobmanager
spec:
  type: ClusterIP
  ports:
  - name: rpc
    port: 6123
  - name: blob-server
    port: 6124
  - name: webui
    port: 8081
  selector:
    app: flink
    component: jobmanager
```

jobmanager-session-deployment.yaml


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flink-jobmanager
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flink
      component: jobmanager
  template:
    metadata:
      labels:
        app: flink
        component: jobmanager
    spec:
      containers:
      - name: jobmanager
        image: flink:1.11.0-scala_2.11
        args: ["jobmanager"]
        ports:
        - containerPort: 6123
          name: rpc
        - containerPort: 6124
          name: blob-server
        - containerPort: 8081
          name: webui
        livenessProbe:
          tcpSocket:
            port: 6123
          initialDelaySeconds: 30
          periodSeconds: 60
        volumeMounts:
        - name: flink-config-volume
          mountPath: /opt/flink/conf
        securityContext:
          runAsUser: 9999  # refers to user _flink_ from official flink image, change if necessary
      volumes:
      - name: flink-config-volume
        configMap:
          name: flink-config
          items:
          - key: flink-conf.yaml
            path: flink-conf.yaml
          - key: log4j-console.properties
            path: log4j-console.properties
```

taskmanager-session-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flink-taskmanager
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flink
      component: taskmanager
  template:
    metadata:
      labels:
        app: flink
        component: taskmanager
    spec:
      containers:
      - name: taskmanager
        image: flink:1.11.0-scala_2.11
        args: ["taskmanager"]
        ports:
```



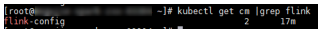
```
    - containerPort: 6122
      name: rpc
    - containerPort: 6125
      name: query-state
    livenessProbe:
      tcpSocket:
        port: 6122
      initialDelaySeconds: 30
      periodSeconds: 60
    volumeMounts:
    - name: flink-config-volume
      mountPath: /opt/flink/conf/
    securityContext:
      runAsUser: 9999  # refers to user _flink_ from official flink image, change if necessary
  volumes:
  - name: flink-config-volume
    configMap:
      name: flink-config
      items:
      - key: flink-conf.yaml
        path: flink-conf.yaml
      - key: log4j-console.properties
        path: log4j-console.properties
```

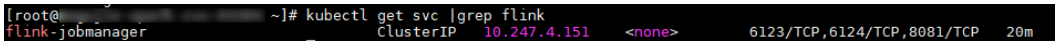
kubectl create -f flink-configuration-configmap.yaml

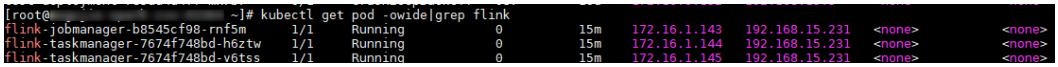
kubectl create -f jobmanager-service.yaml

kubectl create -f jobmanager-session-deployment.yaml

kubectl create -f taskmanager-session-deployment.yaml

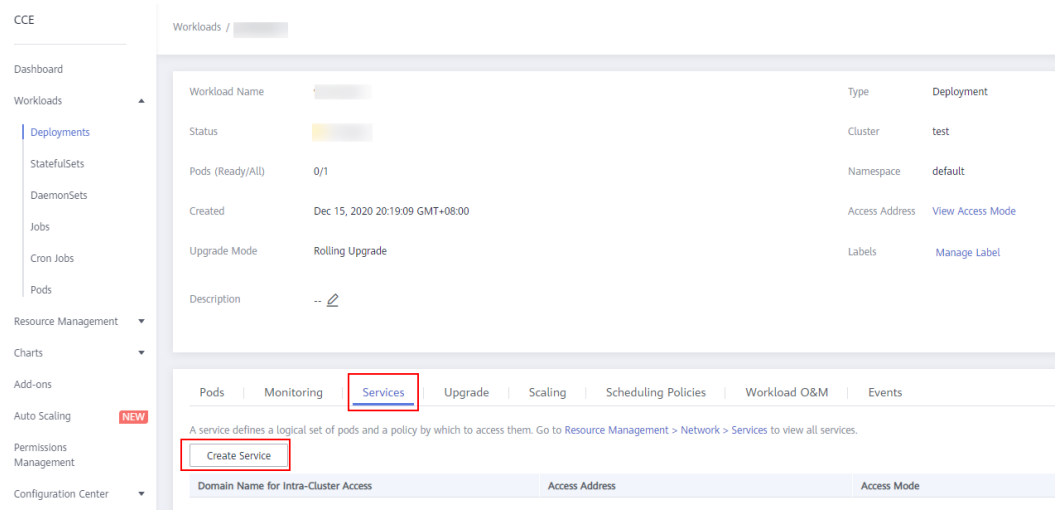




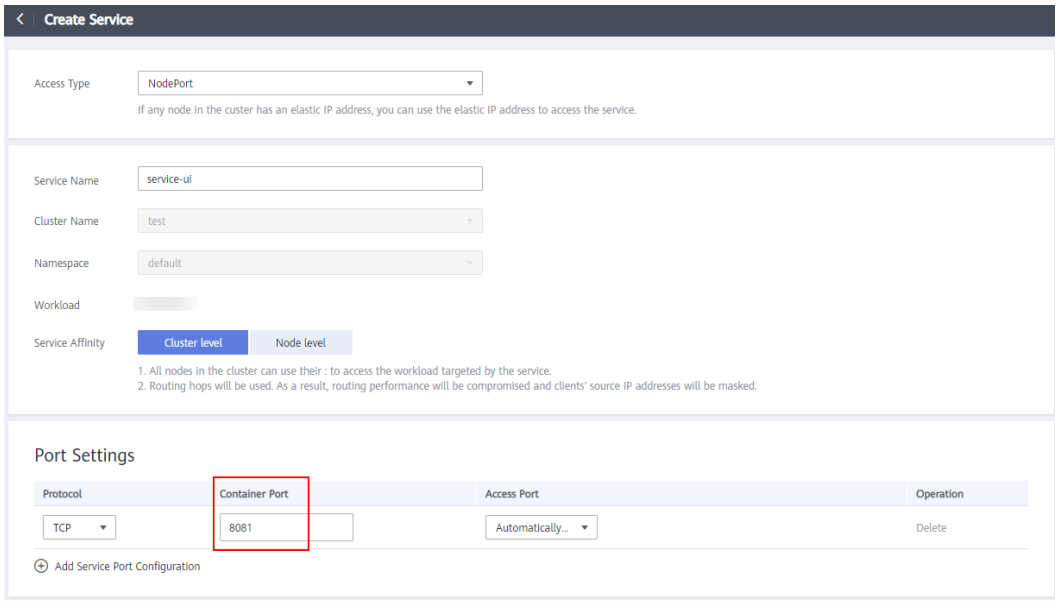


Exposing the Service

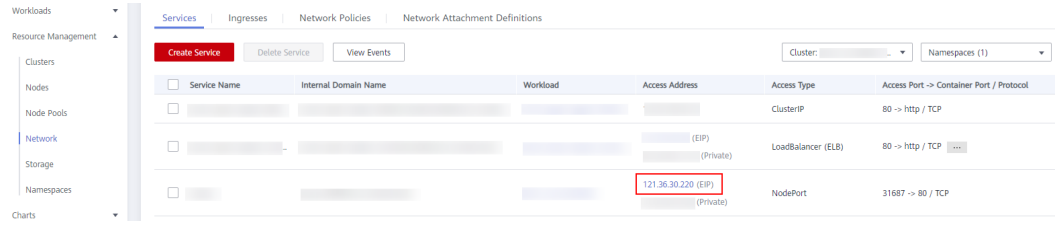
Log in to the CCE console. Choose **Workloads** > **Deployments**, click **flink-jobmanager**, and click the **Services** tab.



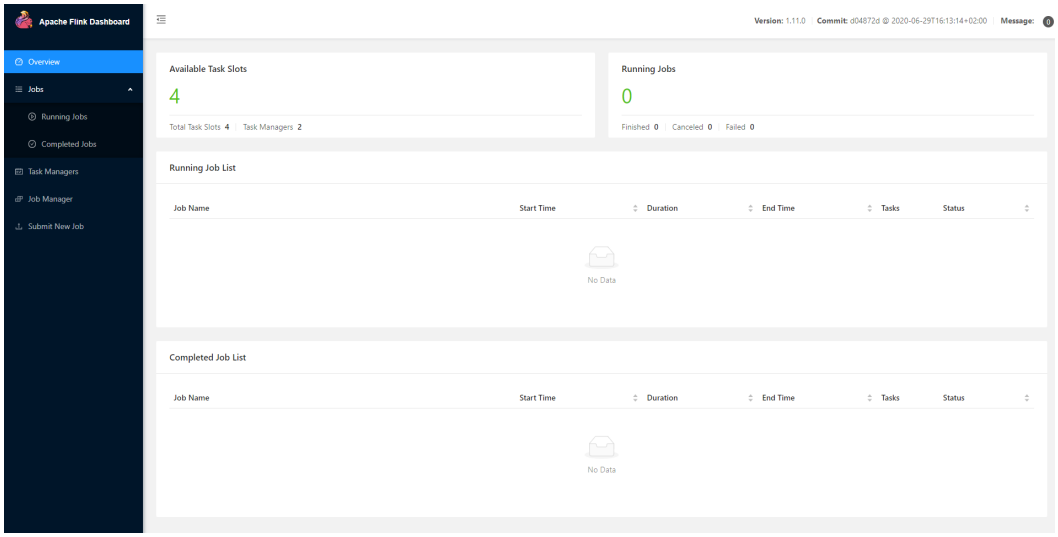
Click **Create Service**, select **NodePort** for **Access Type**, and set **Container Port** to **8081**.



Check whether the Flink can be accessed by using the access address of the Service.



The **Apache Flink Dashboard** page is displayed.

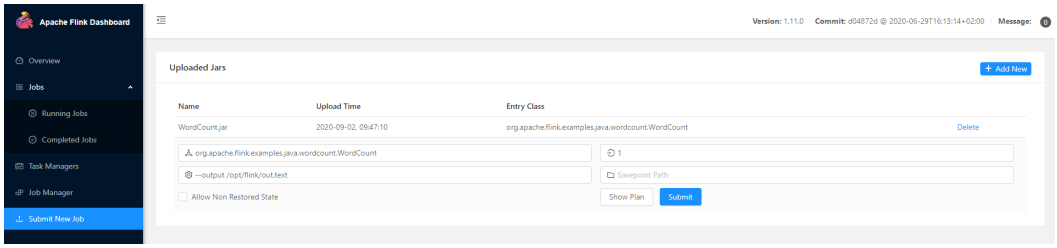


Running the Flink Job

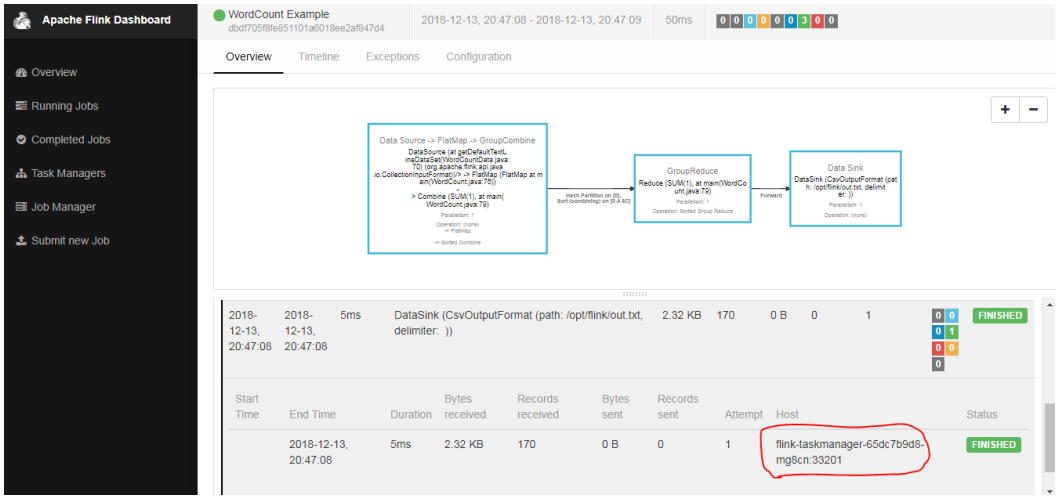
Use the official file **WordCount.jar** to run the Flink job.

Download **https://archive.apache.org/dist/flink/flink-1.11.0/flink-1.11.0-bin-scala_2.11.tgz** and decompress it. The **WordCount.jar** package is generated in **examples\streaming**.

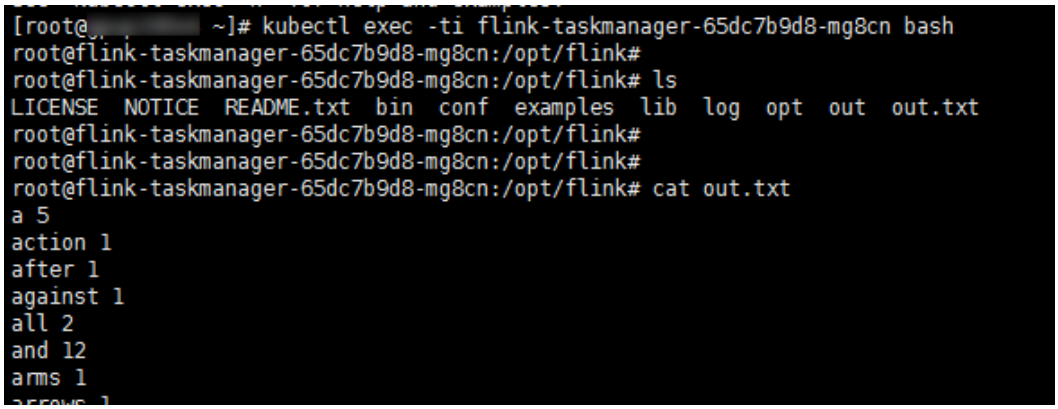
Upload **WordCount.jar**, and set the parameters as follows:



Run the job.



After the job execution is complete, check the execution result. Access the specified task manager and check whether the count of each word is correctly recorded in the **/opt/flink/out** file.



15.5 Deploying ClickHouse on CCE

15.5.1 Planning Resources

ClickHouse Operator creates, configures, and manages ClickHouse clusters running on Kubernetes.

This section describes how to install and deploy ClickHouse Operator on CCE clusters and provides examples of creating ClickHouse cluster resources. For details, see **https://github.com/Altinity/clickhouse-operator**.

ClickHouse Operator can be installed in CCE clusters of v1.15.11 and later. In this example, ClickHouse Operator is installed in a cluster of v1.19.

Cluster type	CCE cluster
Cluster version	1.19
Region	CN East-Shanghai1
Docker version	18.09.0.91
Network model	VPC network
Service forwarding mode	iptables

15.5.2 Configuring kubectl

kubectl is a command line tool for Kubernetes clusters. You can install kubectl on any node and run kubectl commands to perform operations on your Kubernetes cluster.

For details about how to install kubectl, see **Connecting to a Cluster Using kubectl**. After connection, run the **kubectl cluster-info** command to view the cluster information. The following shows an example:

```
# kubectl cluster-info
Kubernetes master is running at https://*.*.*.*:5443
CoreDNS is running at https://*.*.*.*:5443/api/v1/namespaces/kube-system/services/coredns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

15.5.3 Deploying ClickHouse Operator

kubectl apply -f https://github.com/Altinity/clickhouse-operator/blob/master/deploy/operator/clickhouse-operator-install-bundle.yaml

```
[root@click-house-49966 ~]# kubectl apply -f https://raw.githubusercontent.com/Altinity/clickhouse-operator/master/deploy/operator/clickhouse-operator-install.yaml
customresourcedefinition.apiextensions.k8s.io/clickhouseinstallations.clickhouse.altinity.com created
customresourcedefinition.apiextensions.k8s.io/clickhouseinstallationtemplates.clickhouse.altinity.com created
customresourcedefinition.apiextensions.k8s.io/clickhouseoperatorconfigurations.clickhouse.altinity.com created
serviceaccount/clickhouse-operator created
clusterrolebinding.rbac.authorization.k8s.io/clickhouse-operator-kube-system created
configmap/etc-clickhouse-operator-files created
configmap/etc-clickhouse-operator-confd-files created
configmap/etc-clickhouse-operator-configd-files created
configmap/etc-clickhouse-operator-templatesd-files created
configmap/etc-clickhouse-operator-usersd-files created
deployment.apps/clickhouse-operator created
service/clickhouse-operator-metrics created
```

After a period of time, check the running status of ClickHouse Operator.

kubectl get pod -n kube-system|grep clickhouse

```
[root@click-house-49966 ~]# kubectl get pod -n kube-system|grep clickhouse
clickhouse-operator-5d56f7b4b7-zmp98      2/2     Running   0          16m
```


15.5.4 Example

Creating a Namespace

Create a namespace named **test-clickhouse-operator** to facilitate verification of basic functions.

kubectl create namespace test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl create namespace test-clickhouse-operator
namespace/test-clickhouse-operator created
```

Simple Example

This example is available at **https://github.com/Altinity/clickhouse-operator/blob/master/docs/chi-examples/01-simple-layout-01-1shard-1repl.yaml**.

The YAML file is as follows:

```
apiVersion: "clickhouse.altinity.com/v1"
kind: "ClickHouseInstallation"
metadata:
  name: "simple-01"
```

Run the following command:

kubectl apply -n test-clickhouse-operator -f https://raw.githubusercontent.com/Altinity/clickhouse-operator/master/docs/chi-examples/01-simple-layout-01-1shard-1repl.yaml

```
[root@click-house-49966 ~]# kubectl apply -n test-clickhouse-operator -f https://raw.githubusercontent.com/Altinity/clickhouse-operator/master/docs/chi-examples/01-simple-layout-01-1shard-1repl.yaml
clickhouseinstallation.clickhouse.altinity.com/simple-01 created
```

After a period of time, check the resource running status.

kubectl get pod -n test-clickhouse-operator

kubectl get service -n test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl get pod -n test-clickhouse-operator
NAME                          READY   STATUS    RESTARTS   AGE
chi-simple-01-cluster-0-0-0   1/1     Running   0          2m27s
[root@click-house-49966 ~]# kubectl get service -n test-clickhouse-operator
NAME                          TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                       AGE
chi-simple-01-cluster-0-0     ClusterIP      None           <none>        8123/TCP,9000/TCP,9009/TCP    2m3s
clickhouse-simple-01          LoadBalancer   10.247.1.133   <pending>     8123:30485/TCP,9000:30301/TCP   2m55s
```

Connect to the ClickHouse database.

kubectl -n test-clickhouse-operator exec -ti chi-simple-01-cluster-0-0-0 -- clickhouse-client

```
[root@click-house-49966 ~]# kubectl -n test-clickhouse-operator exec -ti chi-simple-01-cluster-0-0-0 -- clickhouse-client
ClickHouse client version 20.8.2.3 (official build).
Connecting to localhost:9000 as user default.
Connected to ClickHouse server version 20.8.2 revision 54438.

chi-simple-01-cluster-0-0-0.chi-simple-01-cluster-0-0.test-clickhouse-operator.svc.cluster.local :)
```

Simple Persistent Volume Example

This example is available at **https://github.com/Altinity/clickhouse-operator/blob/master/docs/chi-examples/03-persistent-volume-01-default-volume.yaml**.

Before using this YAML file to create a PVC on CCE, modify the file based on the storage volume you want to use.

- If an EVS disk is used as a storage volume, do as follows:
  a. Create a StorageClass.

     By default, the CSI disk type supported by CCE is SAS. If you want to use ultra-high I/O EVS disks, create the corresponding StorageClass.

     vim csi-disk-ssd.yaml

     Copy the following content:

     ```
     allowVolumeExpansion: true
     apiVersion: storage.k8s.io/v1
     kind: StorageClass
     metadata:
       name: csi-disk-ssd
     parameters:
       csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
       csi.storage.k8s.io/fstype: ext4
       everest.io/disk-volume-type: SSD
       everest.io/passthrough: "true"
     provisioner: everest-csi-provisioner
     reclaimPolicy: Delete
     volumeBindingMode: Immediate
     ```

     Save the file and exit.

     kubectl create -f csi-disk-ssd.yaml

  b. Set **accessModes** to **ReadWriteOnce**.
  c. Add **storageClassName: csi-disk-ssd**.
- If an SFS file system is used as a storage volume, do as follows:
  a. Set **accessModes** to **ReadWriteMany**.
  b. Add **storageClassName: csi-nas**.

  For example, if an SFS file system is used, the YAML file content is as follows:

  ```
  apiVersion: "clickhouse.altinity.com/v1"
  kind: "ClickHouseInstallation"
  metadata:
    name: "pv-simple"
  spec:
    defaults:
      templates:
        dataVolumeClaimTemplate: data-volume-template
        logVolumeClaimTemplate: log-volume-template
    configuration:
      clusters:
        - name: "simple"
          layout:
            shardsCount: 1
            replicasCount: 1
    templates:
      volumeClaimTemplates:
        - name: data-volume-template
          spec:
            accessModes:
              - ReadWriteMany
            resources:
              requests:
                storage: 10Gi
            storageClassName: csi-nas
        - name: log-volume-template
          spec:
            accessModes:
              - ReadWriteMany
  ```



```
      resources:
        requests:
          storage: 10Gi
      storageClassName: csi-nas
```

Run the following command to create a PV:

kubectl -n test-clickhouse-operator create -f 03-persistent-volume-01-default-volume.yaml

```
[root@click-house-49966 ~]# kubectl -n test-clickhouse-operator create -f 03-persistent-volume-01-default-volume.yaml
clickhouseinstallation.clickhouse.altinity.com/pv-simple created
```

After a period of time, check the resource running status.

kubectl get pvc -n test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl get pvc -n test-clickhouse-operator
NAME                                          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
data-volume-template-chi-pv-simple-simple-0-0-0   Bound    pvc-b3f93d71-ee6a-4d19-96f4-cc93ed69a023   10Gi       RWX            csi-nas        28s
log-volume-template-chi-pv-simple-simple-0-0-0    Bound    pvc-a227b285-e955-414c-bb7f-89b7400bb0fa   10Gi       RWX            csi-nas        28s
```

kubectl get pod -n test-clickhouse-operator

```
[root@click-house-49966 ~]# kubectl get pod -n test-clickhouse-operator
NAME                                 READY   STATUS    RESTARTS   AGE
chi-pv-simple-simple-0-0-0           2/2     Running   0          4m10s
chi-simple-01-cluster-0-0-0          1/1     Running   0          4h11m
```

Run the following command to check the mounting status of the storage volume:

kubectl -n test-clickhouse-operator exec -ti chi-pv-simple-simple-0-0-0 -c clickhouse bash

df -h

```
[root@click-house-49966 ~]# kubectl -n test-clickhouse-operator exec -ti chi-pv-simple-simple-0-0-0 -c clickhouse bash
root@chi-pv-simple-simple-0-0-0:/#
root@chi-pv-simple-simple-0-0-0:/# df -h
Filesystem                                                      Size  Used Avail Use% Mounted on
/dev/mapper/docker-252:1-262145-23a1e3cdc4dfeca71936bcd6762a859a7348817e20e0d2101fc12ea8fb420201  9.8G  707M  8.6G   8% /
tmpfs                                                            64M     0   64M   0% /dev
tmpfs                                                           7.8G     0  7.8G   0% /sys/fs/cgroup
/dev/mapper/vgpaas-kubernetes                                   9.8G   37M  9.2G   1% /etc/hosts
/dev/mapper/vgpaas-dockersys                                     18G   90M   17G   1% /etc/hostname
shm                                                              64M     0   64M   0% /dev/shm
sfs-nas01.cn-east-3a.myhuaweicloud.com:/share-9050edca           10G  1.0M   10G   1% /var/lib/clickhouse
sfs-nas01.cn-east-3a.myhuaweicloud.com:/share-5bc94443           10G     0   10G   0% /var/log/clickhouse-server
tmpfs                                                           7.8G   12K  7.8G   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs                                                           7.8G     0  7.8G   0% /proc/acpi
tmpfs                                                           7.8G     0  7.8G   0% /proc/scsi
tmpfs                                                           7.8G     0  7.8G   0% /sys/firmware
```

Connect to the ClickHouse database.

kubectl -n test-clickhouse-operator exec -ti chi-pv-simple-simple-0-0-0 -- clickhouse-client

```
[root@click-house-49966 ~]# kubectl -n test-clickhouse-operator exec -ti chi-pv-simple-simple-0-0-0 -- clickhouse-client
Defaulting container name to clickhouse.
Use 'kubectl describe pod/chi-pv-simple-simple-0-0-0 -n test-clickhouse-operator' to see all of the containers in this pod.
ClickHouse client version 20.8.2.3 (official build).
Connecting to localhost:9000 as user default.
Connected to ClickHouse server version 20.8.2 revision 54438.

chi-pv-simple-simple-0-0-0.chi-pv-simple-simple-0-0.test-clickhouse-operator.svc.cluster.local :)
```

Simple Load Balancer Example

```
apiVersion: "clickhouse.altinity.com/v1"
kind: "ClickHouseInstallation"
metadata:
  name: "ck-elb"
spec:
  defaults:
    templates:
      dataVolumeClaimTemplate: data-volume-nas
      serviceTemplate: chi-service-elb
  configuration:
    clusters:
      - name: "ck-elb"
        templates:
          podTemplate: pod-template-with-nas
        layout:
          shardsCount: 1
          replicasCount: 1
  templates:
```



```
  podTemplates:
    - name: pod-template-with-nas
      spec:
        containers:
          - name: clickhouse
            image: yandex/clickhouse-server:21.6.3.14
            volumeMounts:
              - name: data-volume-nas
                mountPath: /var/lib/clickhouse
  volumeClaimTemplates:
    - name: data-volume-nas
      spec:
        accessModes:
          - ReadWriteMany
        resources:
          requests:
            storage: 20Gi
        storageClassName: csi-nas
  serviceTemplates:
    - name: chi-service-elb
      metadata:
        annotations:
          kubernetes.io/elb.class: union
          kubernetes.io/elb.autocreate: >-
            {"type":"public","bandwidth_name":"cce-bandwidth-
ck","bandwidth_chargemode":"bandwidth","bandwidth_size":5,"bandwidth_sharetype":"PER","eip_typ
e":"5_bgp"}
        spec:
          ports:
            - name: http
              port: 8123
            - name: client
              port: 9000
          type: LoadBalancer
```

Add the information in bold to the YAML file. The following table describes the parameters supported by **annotations kubernetes.io/elb.autocreate**.

Parameter	Type	Description
name	String	Name of the automatically created load balancer. Value range: a string of 1 to 64 characters, including lowercase letters, digits, and underscores (_). The value must start with a lowercase letter and end with a lowercase letter or digit.
type	String	Network type of the load balancer. • **public**: public network load balancer • **inner**: private network load balancer
bandwidth_name	String	Bandwidth name. The default value is **cce-bandwidth-******. Value range: a string of 1 to 64 characters, including lowercase letters, digits, and underscores (_). The value must start with a lowercase letter and end with a lowercase letter or digit.

Parameter	Type	Description
bandwidth_chargemode	String	Bandwidth billing mode. • **bandwidth**: billed by bandwidth • **traffic**: billed by traffic
bandwidth_size	Integer	Bandwidth.
bandwidth_sharetype	String	Bandwidth sharing mode. • **PER**: dedicated bandwidth
eip_type	String	EIP type.

15.6 Running Spark on CCE

15.6.1 Installing Spark

Prerequisites

A Linux server that can access the public network is available. The recommended node specifications are 4 vCPUs and 8 GiB memory or higher.

Configuring the JDK

This section uses CentOS as an example to describe how to install JDK 1.8.

Step 1 Obtain the available version.

```
yum -y list java*
```

Step 2 Install JDK 1.8.

```
yum install -y java-1.8.0-openjdk java-1.8.0-openjdk-devel
```

Step 3 Check the version after the installation.

```
# java -version
openjdk version "1.8.0_382"
OpenJDK Runtime Environment (build 1.8.0_382-b05)
OpenJDK 64-Bit Server VM (build 25.382-b05, mixed mode)
```

Step 4 Add environment variables.

1. Linux environment variables are configured in the **/etc/profile** file.
   ```
   vim /etc/profile
   ```
2. In the editing mode, add the following content to the end of the file:
   ```
   JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.382.b05-1.el7_9.x86_64
   PATH=$PATH:$JAVA_HOME/bin
   CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
   export JAVA_HOME  PATH  CLASSPATH
   ```
3. Save and close the **profile** file. Run the following command for the modification to take effect:
   ```
   source /etc/profile
   ```
4. Check the JDK environment variables.


```
echo $JAVA_HOME
echo $PATH
echo $CLASSPATH
```

----End

Obtaining the Spark Package

OBS matches Hadoop 2.8.3 and 3.1.1. Hadoop 3.1.1 is used in this example.

Step 1 Download Spark v3.1.1. If Git is not installed, run **yum install git** to install it.

```
git clone -b v3.1.1 https://github.com/apache/spark.git
```

Step 2 Modify the **/dev/make-distribution.sh** file and specify the Spark version so that the check can be skipped during compilation.

1. Search for the line where **VERSION** resides and check the number of the line where the version number is located.
   ```
   cat ./spark/dev/make-distribution.sh |grep -n '^VERSION=' -A18
   ```
2. Comment out the content displayed from lines 129 to 147 and specify the version.
   ```
   sed -i '129,147s/^/#/g' ./spark/dev/make-distribution.sh
   sed -i '148a
   VERSION=3.1.3\nSCALA_VERSION=2.12\nSPARK_HADOOP_VERSION=3.1.1\nSPARK_HIVE=1' ./
   spark/dev/make-distribution.sh
   ```

Step 3 Download the dependency.

```
wget https://archive.apache.org/dist//maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
tar -zxvf apache-maven-3.6.3-bin.tar.gz && mv apache-maven-3.6.3 ./spark/build
```

Step 4 Run the following command to perform compilation:

```
./spark/dev/make-distribution.sh --name hadoop3.1 --tgz -Pkubernetes -Pyarn -Dhadoop.version=3.1.1
```

Step 5 Wait for the compilation to complete. After the compilation, the software package is named **spark-3.1.3-bin-hadoop3.1**.tgz.

----End

Configuring the Runtime Environment for Spark

To simplify the operation, use the **root** user to place the compiled package **spark-3.1.3-bin-hadoop3.1.tgz** in the **/root** directory on the operation node.

Step 1 Move the software package to the **/root** directory.

```
mv ./spark/spark-3.1.3-bin-hadoop3.1.tgz /root
```

Step 2 Run the following command to install Spark:

```
tar -zxvf spark-3.1.3-bin-hadoop3.1.tgz
mv spark-3.1.3-bin-hadoop3.1 spark-obs
cat >> ~/.bashrc <<EOF
PATH=/root/spark-obs/bin:\$PATH
PATH=/root/spark-obs/sbin:\$PATH
export SPARK_HOME=/root/spark-obs
EOF

source ~/.bashrc
```

Step 3 Run the following command where binary **spark-submit** is used to check the Spark version:

```
spark-submit --version
```

----End

Interconnecting Spark with OBS

Step 1 Obtain Huawei Cloud OBS JAR. The **hadoop-huaweicloud-3.1.1-hw-45.jar** package is used, which can be obtained from **https://github.com/huaweicloud/obsa-hdfs/tree/master/release**.

```
wget https://github.com/huaweicloud/obsa-hdfs/releases/download/v45/hadoop-huaweicloud-3.1.1-hw-45.jar
```

Step 2 Copy the package to the corresponding directory.

```
cp hadoop-huaweicloud-3.1.1-hw-45.jar /root/spark-obs/jars/
```

Step 3 Modify Spark configuration items. To interconnect Spark with OBS, add ConfigMaps for Spark as follows:

1. Obtain the AK/SK. For details, see **Access Keys**.
2. Change the values of **AK_OF_YOUR_ACCOUNT**, **SK_OF_YOUR_ACCOUNT**, and **OBS_ENDPOINT** to the actual values.
   - **AK_OF_YOUR_ACCOUNT**: indicates the AK obtained in the previous step.
   - **SK_OF_YOUR_ACCOUNT**: indicates the SK obtained in the previous step.
   - **OBS_ENDPOINT**: indicates the OBS endpoint. It can be obtained in **Regions and Endpoints**.

```
cp ~/spark-obs/conf/spark-defaults.conf.template ~/spark-obs/conf/spark-defaults.conf

cat >> ~/spark-obs/conf/spark-defaults.conf <<EOF
spark.hadoop.fs.obs.readahead.inputstream.enabled=true
spark.hadoop.fs.obs.buffer.max.range=6291456
spark.hadoop.fs.obs.buffer.part.size=2097152
spark.hadoop.fs.obs.threads.read.core=500
spark.hadoop.fs.obs.threads.read.max=1000
spark.hadoop.fs.obs.write.buffer.size=8192
spark.hadoop.fs.obs.read.buffer.size=8192
spark.hadoop.fs.obs.connection.maximum=1000
spark.hadoop.fs.obs.access.key=AK_OF_YOUR_ACCOUNT
spark.hadoop.fs.obs.secret.key=SK_OF_YOUR_ACCOUNT
spark.hadoop.fs.obs.endpoint=OBS_ENDPOINT
spark.hadoop.fs.obs.buffer.dir=/root/hadoop-obs/obs-cache
spark.hadoop.fs.obs.impl=org.apache.hadoop.fs.obs.OBSFileSystem
spark.hadoop.fs.obs.connection.ssl.enabled=false
spark.hadoop.fs.obs.fast.upload=true
spark.hadoop.fs.obs.socket.send.buffer=65536
spark.hadoop.fs.obs.socket.recv.buffer=65536
spark.hadoop.fs.obs.max.total.tasks=20
spark.hadoop.fs.obs.threads.max=20
spark.kubernetes.container.image.pullSecrets=default-secret
EOF
```

----End

Pushing an Image to SWR

To run Spark tasks in Kubernetes, build a Spark container image of the same version and upload it to SWR. A **Dockerfile** file has been generated during compilation. Use this file to create an image and push it to SWR.

Step 1 Create an image.

```
cd ~/spark-obs
docker build -t spark:3.1.3-obs --build-arg spark_uid=0 -f kubernetes/dockerfiles/spark/Dockerfile .
```

Step 2 Upload the image.

1. (Optional) Log in to the SWR console, choose **Organizations** in the navigation pane, and click **Create Organization** in the upper right corner of the page.

   Skip this step if you already have an organization.

2. Choose **My Images** in the navigation pane and click **Upload Through Client**. On the page displayed, click **Generate a temporary login command** and click  to copy the command.

3. Run the login command copied in the previous step on the node. If the login is successful, the message "Login Succeeded" will display.

4. Log in to the node where the image is created and run the login command.

   ```
   Docker tag [{ Image name }:{Version name }] swr. ap-southeast-1.myhuaweicloud.com/{ Organization name }/{Image name }:{version name }
   docker push swr.ap-southeast-1.myhuaweicloud.com/{Organization name }/{Image name }:{Version name }
   ```

   Record the image access address for later use.

   For example, record the IP address as **swr.ap-southeast-1.myhuaweicloud.com/dev-container/spark:3.1.3-obs**.

----End

Configuring Spark History Server

Step 1 Modify the **~/spark-obs/conf/spark-defaults.conf** file, enable Spark event logging, and configure the OBS bucket name and directory.

```
cat >> ~/spark-obs/conf/spark-defaults.conf <<EOF
spark.eventLog.enabled=true
spark.eventLog.dir=obs://{bucket-name}/{log-dir}/
EOF
```

- **spark.eventLog.enabled**: indicates that Spark event logging is enabled if it is set to **true**.
- **spark.eventLog.dir**: indicates the OBS bucket name and path. The bucket is named in the format of **obs://***{bucket-name}***/***{log-dir}***/**, for example, **obs://spark-sh1/history-obs/**. Ensure that the OBS bucket name and directory are correct.

Step 2 Modify the **~/spark-obs/conf/spark-env.sh** file. If the file does not exist, run the command to copy the template as a file:

```
cp ~/spark-obs/conf/spark-env.sh.template ~/spark-obs/conf/spark-env.sh

cat >> ~/spark-obs/conf/spark-env.sh <<EOF
SPARK_HISTORY_OPTS="-Dspark.history.fs.logDirectory=obs://{bucket-name}/{log-dir}/"
EOF
```

The OBS address must be the same as that in **spark-default.conf** in the previous step.

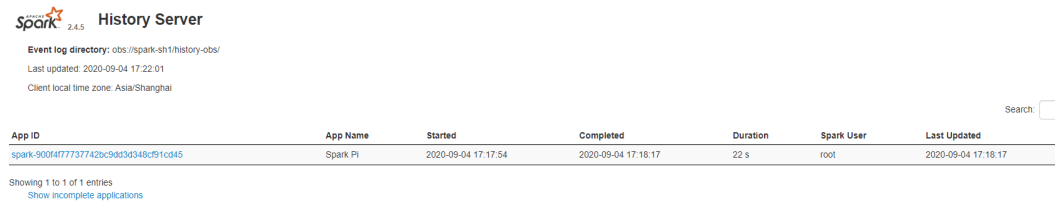
Step 3 Start Spark History Server.

```
start-history-server.sh
```

Information similar to the following is displayed:

```
starting org.apache.spark.deploy.history.HistoryServer, logging to /root/spark-obs/logs/spark-root-org.apache.spark.deploy.history.HistoryServer-1-spark-sh1.out
```

Step 4 Access the server through port 18080 on the node.



To stop the server, run the following command:

```
stop-history-server.sh
```

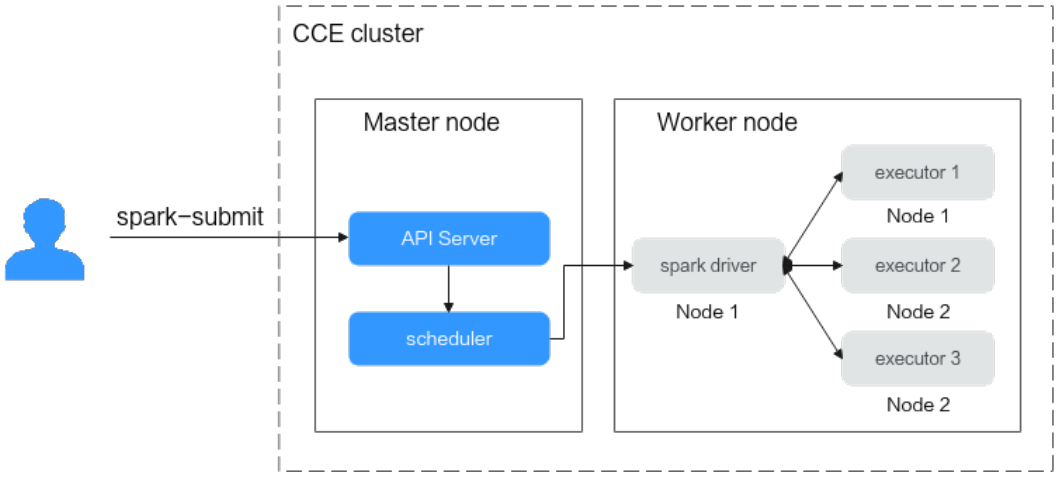
----End

15.6.2 Using Spark on CCE

You can use the Kubernetes scheduler spark-submit of Spark to submit Spark applications to the Kubernetes clusters. For details, see **Running Spark on Kubernetes**. The submission mechanism works as follows:

- Create a pod to run the Spark driver.
- The driver creates pods for executing the programs and establishes a connection with these pods.
- After the application is complete, the pods that execute the programs are terminated and cleaned up, but the driver pod exists and remains in the completed state until the garbage is collected or it is manually cleaned up. In the completed state, the driver pod does not use any computing or memory resources.

Figure 15-2 Submission mechanism



Running SparkPi on CCE

Step 1 Install kubectl on the node where Spark is running. For details, see **Connecting to a Cluster Using kubectl**.

Step 2 Run the following command to grant the cluster-level permissions:

```
# Create a service account.
kubectl create serviceaccount spark
# Bind the ClusterRole spark-role to the service account created in the previous step, specify the default namespace, and grant the ClusterRole permission to edit resources.
```



```
kubectl create clusterrolebinding spark-role --clusterrole=edit --serviceaccount=default:spark --
namespace=default
```

Step 3 Submit a SparkPi job to CCE. The following shows an example:

```
spark-submit \
  --master k8s://https://**.**.**.**:5443 \
  --deploy-mode cluster \
  --name spark-pi \
  --class org.apache.spark.examples.SparkPi \
  --conf spark.executor.instances=2 \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
  --conf spark.kubernetes.container.image=swr.ap-southeast-1.myhuaweicloud.com/dev-container/
spark:3.1.3-obs \
  local:///root/spark-obs/examples/jars/spark-examples_2.12-3.1.1.jar
```

Parameters:

- **--master**: indicates the API Server of the cluster. **https://**.**.**.***:5443** is the address of the master node used in **~/.kube/config**. It can be obtained from **kubectl cluster-info**.
- **--deploy-mode**:
  - **cluster**: a mode in which the driver is deployed on the worker nodes.
  - **client**: (default value) a mode in which the driver is deployed locally as an external client.
- **--name**: indicates the name of a job. It is used to name the pods in the cluster.
- **--class**: indicates the applications, for example, **org.apache.spark.examples.SparkPi**.
- **--conf**: indicates the Spark's configuration parameters. It is in the key-value pair format. All parameters that can be specified using **--conf** are read from the **~/spark-obs/conf/spark-defaults.conf** file by default. Therefore, the general configuration can be written to be the default settings, the same way as **Interconnecting Spark with OBS**.
  - **spark.executor.instances**: indicates the number of pods for executing programs.
  - **spark.kubernetes.authenticate.driver.serviceAccountName**: indicates the driver's cluster-level permissions. Select the service account created in **Step 2**.
  - **spark.kubernetes.container.image**: indicates the image path of the image pushed to SWR in **Pushing an Image to SWR**.
- **local**: indicates the path to the JAR packages stored in the local files. In this example, a local file is used to store the JAR packages. The value of this parameter can be **file**, **http**, or **local**. For details, see the **Official Document**.

----End

Accessing OBS

Use **spark-submit** to deliver an HDFS job. Change the value of **obs://bucket-name/filename** at the end of the script to the actual file name of the tenant.

```
spark-submit \
  --master k8s://https://**.**.**.**:5443 \
  --deploy-mode cluster \
  --name spark-hdfs-test \
  --class org.apache.spark.examples.HdfsTest \
```



```
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
  --conf spark.kubernetes.container.image=swr.ap-southeast-1.myhuaweicloud.com/dev-container/
spark:3.1.3-obs \
  local:///root/spark-obs/examples/jars/spark-examples_2.12-3.1.1.jar obs://bucket-name/filename
```

Support for Spark Shell Commands to Interact with Spark-Scala

```
spark-shell \
  --conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
  --conf spark.kubernetes.container.image=swr.ap-southeast-1.myhuaweicloud.com/dev-container/
spark:3.1.3-obs \
  --master k8s://https://**.**.**.**:5443
```

Run the following commands to define the algorithms of Spark computing jobs linecount and wordcount:

```
def linecount(input:org.apache.spark.sql.Dataset[String]):Long=input.filter(line => line.length()>0).count()
def wordcount(input:org.apache.spark.sql.Dataset[String]):Long=input.flatMap(value => value.split("\\s
+")).groupByKey(value => value).count().count()
```

Run the following commands to define data sources:

```
var alluxio = spark.read.textFile("alluxio://alluxio-master:19998/sample-1g")
var obs = spark.read.textFile("obs://gene-container-gtest/sample-1g")
var hdfs = spark.read.textFile("hdfs://192.168.1.184:9000/user/hadoop/books/sample-1g")
```

Run the following command to start computing jobs:

```
spark.time(wordcount(obs))
spark.time(linecount(obs))
```

