Cloud Container Engine

Best Practices

Issue 01

Date 2023-09-14

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

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
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.
	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.

Category	Check Item	Type	Impact
	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.
	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.
	When creating a cluster, select a proper network model, such as container tunnel network or VPC network.	Deployment	After a cluster is created, the network model cannot be changed. Exercise caution when selecting a network model.
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.
	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.

Catego ry	Check Item	Type	Impact
	When creating a workload, select a proper access mode (Service). Currently, the following types of Services are supported: ClusterIP, NodePort, and LoadBalancer.	Deploy ment	Improper Service configuration may cause logic confusion for internal and external access and resource waste.
	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.	Reliabi lity	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.
	Properly set affinity and anti-affinity.	Reliabi lity	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.
	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.	Reliabi lity	If the pre-stop processing command is not configured, the pod will be directly killed and services will be interrupted during application upgrade.

Table 1-2 Data reliability

Catego ry	Check Item	Type	Impact
Contain er data persiste ncy	Select a proper data volume type based on service requirements.	Reliabi lity	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.

Category	Check Item	Type	Impact
Backup	Back up application data.	Reliability	Data cannot be restored after being lost.

Table 1-3 O&M reliability

Category	Check Item	Type	Impact
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.
	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.
	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.

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

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 Procedure

2.1.2.1 Containerizing an Entire Application

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

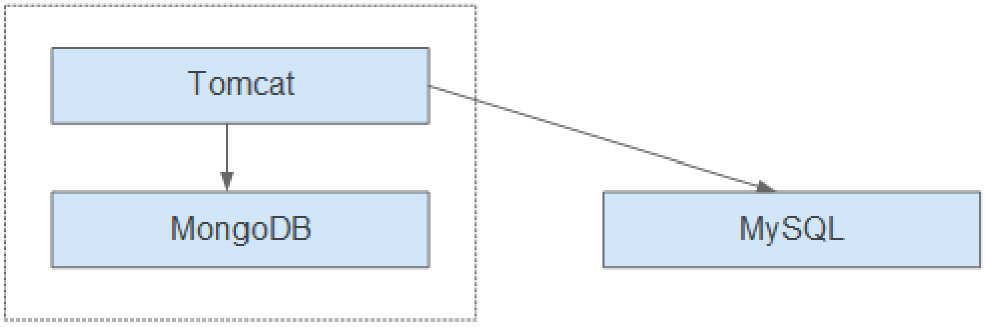
No recoding or re-architecting is required. You only need to pack the entire application into a container image and deploy the container image on CCE.

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.

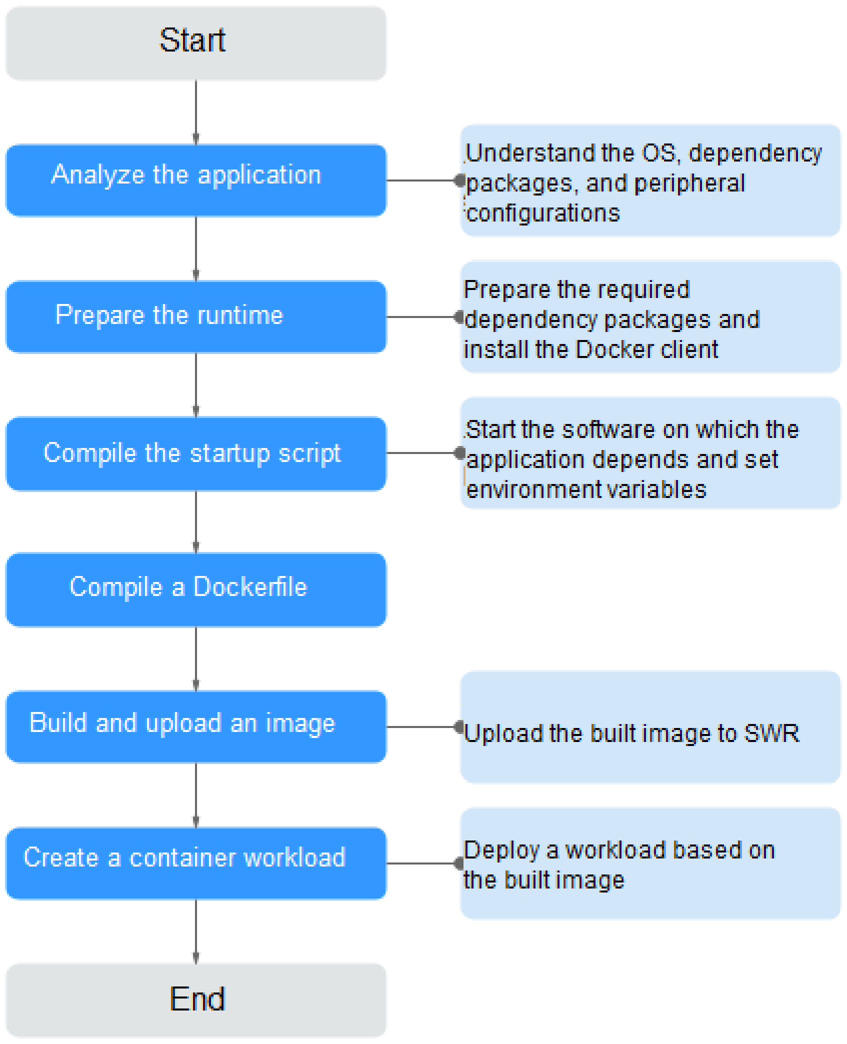
Table 2-2 Comparison between the two deployment modes

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.
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.

2.1.2.2 Containerization Process

The following figure illustrates the process of containerizing an application.

Figure 2-2 Process of containerizing an application


Start
Analyze the application
Understand the OS, dependency packages, and peripheral configurations
Prepare the runtime
Prepare the required dependency packages and install the Docker client
Compile the startup script
Start the software on which the application depends and set environment variables
Compile a Dockerfile
Build and upload an image
Upload the built image to SWR
Create a container workload
Deploy a workload based on the built image
End

2.1.2.3 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 2-3**.

Table 2-3 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`

	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.

2.1.2.4 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.

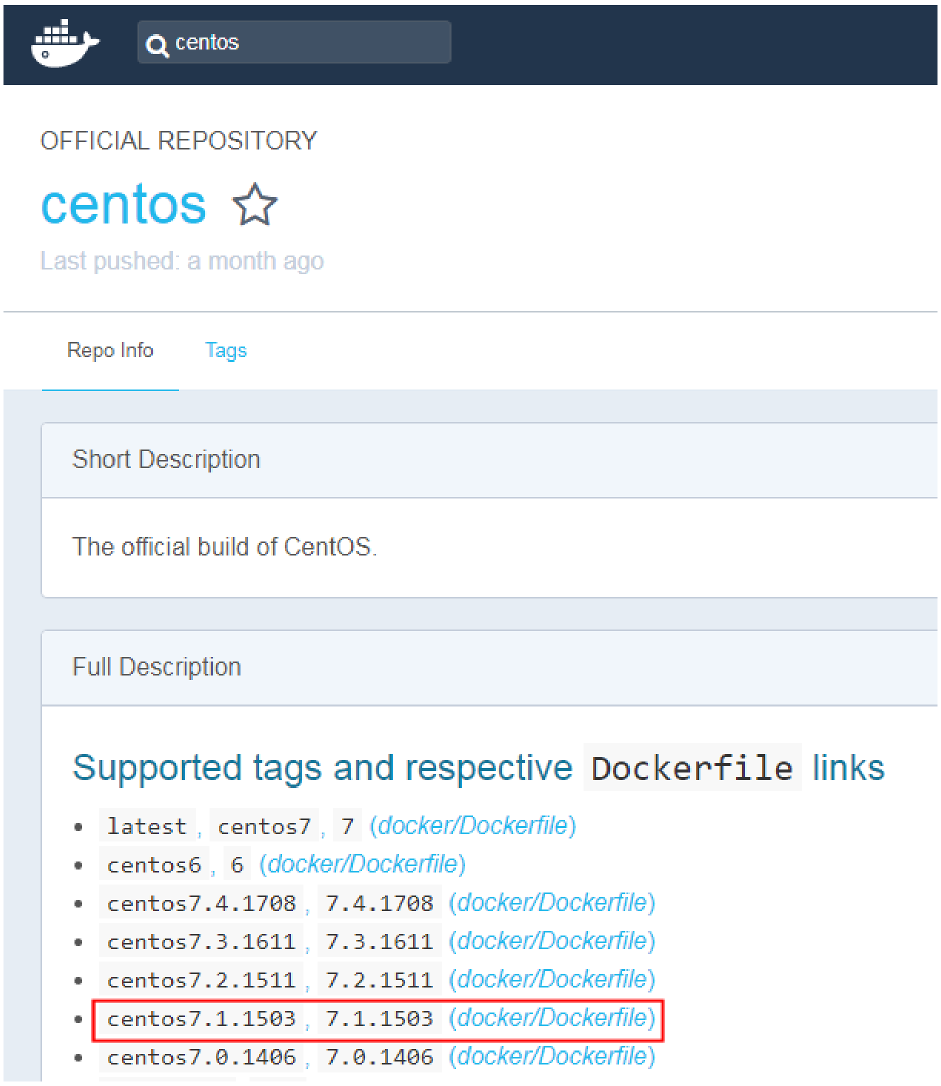
NOTE

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



OFFICIAL REPOSITORY

centos

Last pushed: a month ago

Short Description

The official build of CentOS.

Full Description

Supported tags and respective `Dockerfile` links

- `latest`, `centos7`, `7` (docker/Dockerfile)
- `centos6`, `6` (docker/Dockerfile)
- `centos7.4.1708`, `7.4.1708` (docker/Dockerfile)
- `centos7.3.1611`, `7.3.1611` (docker/Dockerfile)
- `centos7.2.1511`, `7.2.1511` (docker/Dockerfile)
- `centos7.1.1503`, `7.1.1503` (docker/Dockerfile)
- `centos7.0.1406`, `7.0.1406` (docker/Dockerfile)

----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

2.1.2.5 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

2.1.2.6 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.

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

2.1.2.7 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

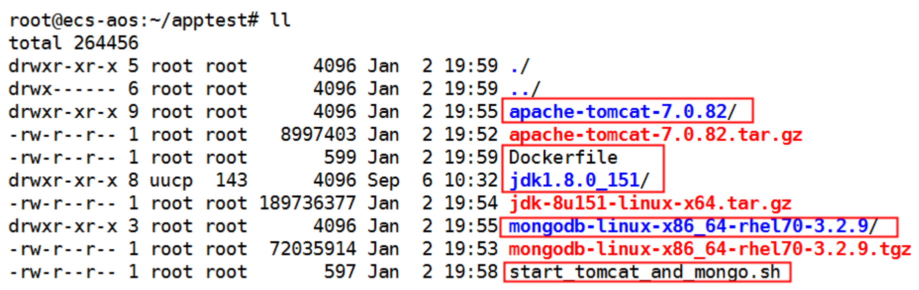
Step 1 Log in as the **root** user to the device running Docker.

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

2.1.2.8 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 2-4**.

Table 2-4 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. Choose **Clusters**. On the displayed page, select the type of the cluster to be created and click **Buy**.

   Configure cluster parameters and select the VPC created in **Step 1**.
2. Buy a node and select the key pair created in **Step 1** as the login mode.

Step 3 Deploy a workload on CCE.

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. 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 2-5 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.

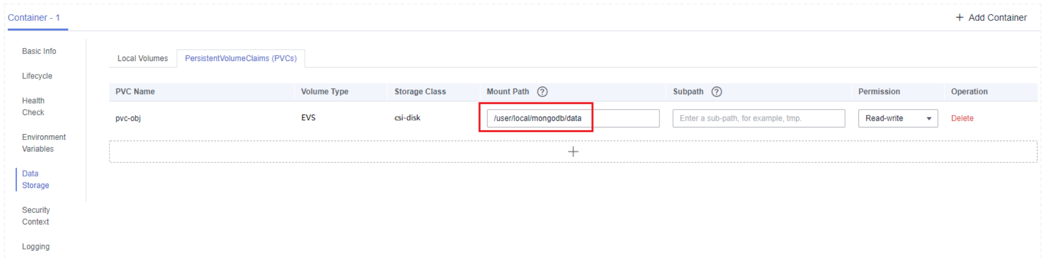
5. In the **Container Settings** area, choose **Data Storage** and configure cloud storage for persistent data storage.

   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



PVC Name	Volume Type	Storage Class	Mount Path	Subpath	Permission	Operation
pvc-obj	EVS	csi-disk	/user/local/mongodb/data	Enter a sub-path, for example, tmp.	Read-write	Delete

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 On-premises Kubernetes Clusters to CCE

3.1.1 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.

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

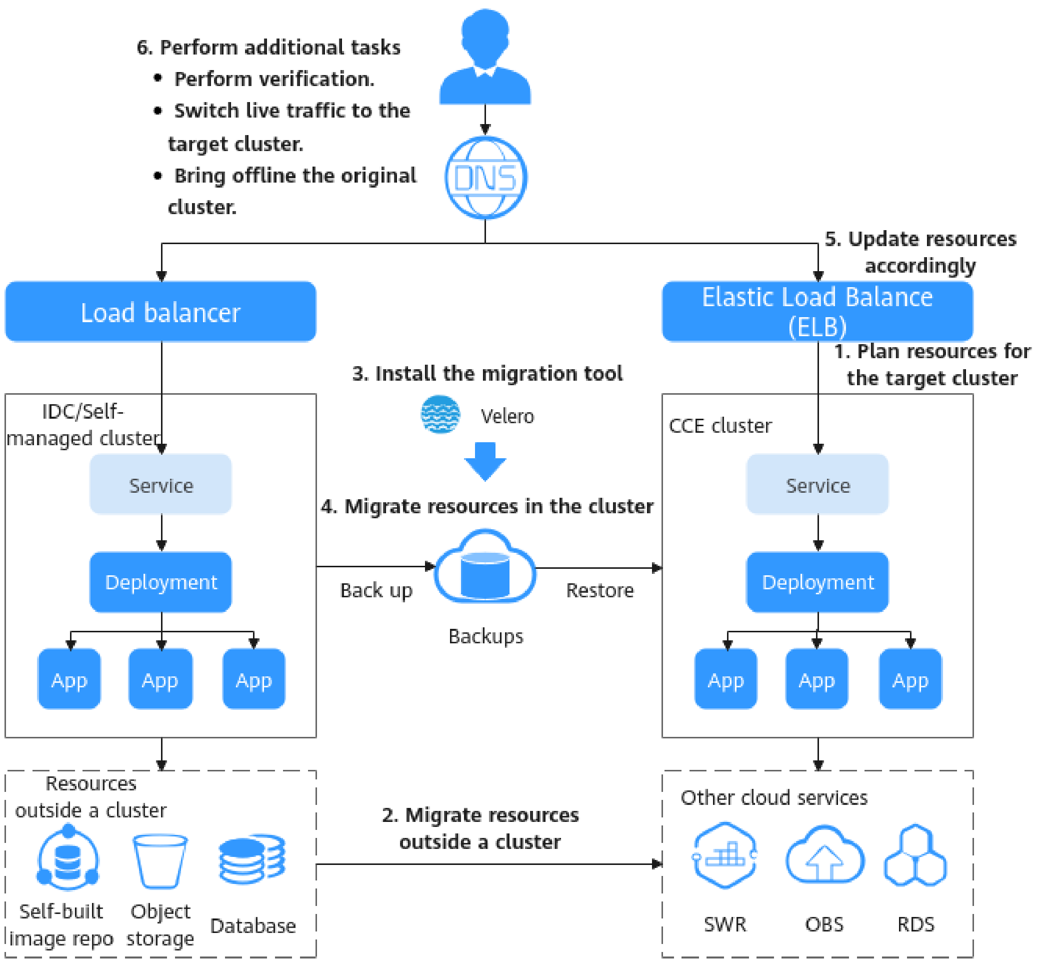
Before the migration, analyze all resources in the source clusters and then determine the migration solution. Resources that can be migrated include resources inside and outside the clusters, as listed in the following table.

Table 3-1 Resources that can be migrated

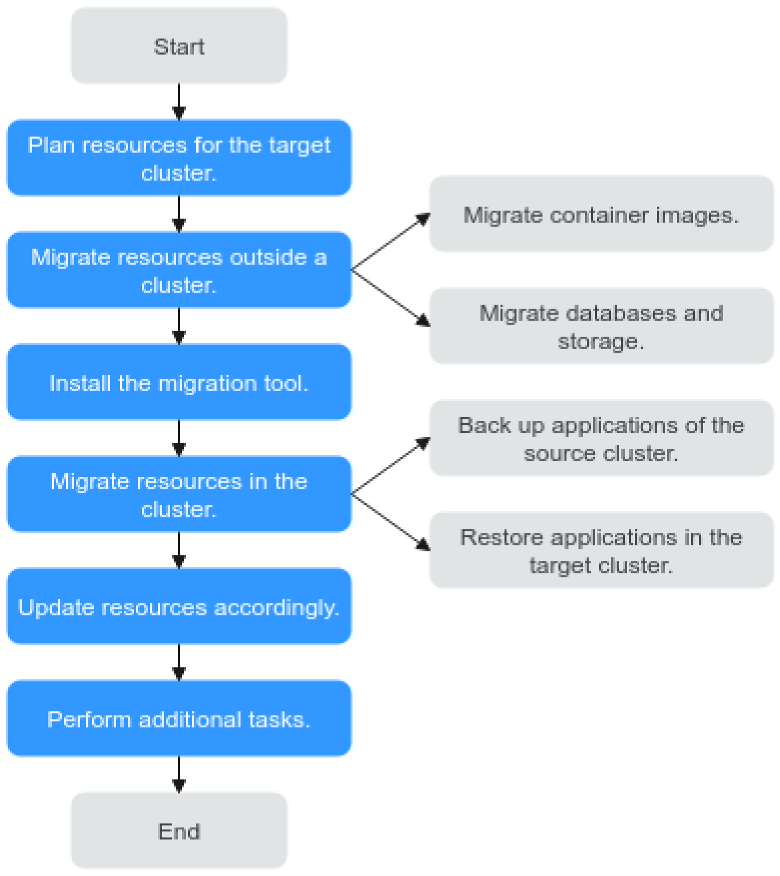
Category	Migration Object	Remarks
Resources inside a cluster	All objects in a cluster, including pods, jobs, Services, Deployments, and ConfigMaps.	You are not advised to migrate the resources in the **velero** and **kube-system** namespaces. ● **velero**: Resources in this namespace are created by the 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.
	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 3-1 shows the migration process. You can migrate resources outside a cluster as required.

Figure 3-1 Migration solution diagram


6. Perform additional tasks
• Perform verification.
• Switch live traffic to the target cluster.
• Bring offline the original cluster.
DNS
5. Update resources accordingly
Load balancer
Elastic Load Balance (ELB)
1. Plan resources for the target cluster
3. Install the migration tool
Velero
IDC/Self-managed cluster
CCE cluster
Service
Deployment
App
App
App
4. Migrate resources in the cluster
Back up
Restore
Backups
Service
Deployment
App
App
App
Resources outside a cluster
Self-built image repo
Object storage
Database
2. Migrate resources outside a cluster
Other cloud services
SWR
OBS
RDS

Migration Process


Start
Plan resources for the target cluster.
Migrate resources outside a cluster.
Migrate container images.
Migrate databases and storage.
Install the migration tool.
Migrate resources in the cluster.
Back up applications of the source cluster.
Restore applications in the target cluster.
Update resources accordingly.
Perform additional tasks.
End

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** .

- **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

3.1.2 Planning Resources for the Target Cluster

CCE allows you to customize cluster resources to meet various service requirements. **Table 3-2** 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 3-2** cannot be modified.

Table 3-2 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
Node	OS	• EulerOS • CentOS	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. • **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. • **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.	General-purpose (node specifications: 4 vCPUs and 8 GiB memory)
	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

3.1.3 Migrating Resources Outside a Cluster

If your migration does not involve resources outside a cluster listed in **Table 3-1** 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:

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:

----End

3.1.4 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.
- 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.

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 the MinIO server or install MinIO in a container. For details, see the official Velero documentation at **https://velero.io/docs/v1.7/contributions/minio/#set-up-server**.

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 4**.

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 Restic 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 1.7.0 as an example. The installation process in the source cluster is the same as that in the target cluster.

Step 1 Download the binary file of Velero 1.7.0.

```
wget https://github.com/vmware-tanzu/velero/releases/download/v1.7.0/velero-v1.7.0-linux-amd64.tar.gz
```

Step 2 Install the Velero client.

```
tar -xvf velero-v1.7.0-linux-amd64.tar.gz
cp ./velero-v1.7.0-linux-amd64/velero /usr/local/bin
```

Step 3 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. 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 4 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**.

Table 3-3 Installation parameters of Velero

Parameter	Description
--provider	Vendor who provides the plug-in.
--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 3**.
--use-restic	Whether to use Restic to support 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 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 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 5 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
restic-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 Restic and Velero by referring to **Backup Tool Resources Are Insufficient**.

Step 6 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

3.1.5 Migrating Resources in a Cluster

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 image 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 integrates the Restic tool to back up and restore storage volumes. Currently, the storage volumes of the HostPath type are not supported. For details, see **Restic Restrictions**. 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-restic**: indicates that the Restic tool is used to back up all storage volumes mounted to the pod. Storage volumes of the HostPath type 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-restic** is specified during **Velero installation**.
  ```
  velero backup create <backup-name> --default-volumes-to-restic
  ```
- **--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-restic
```

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.
```

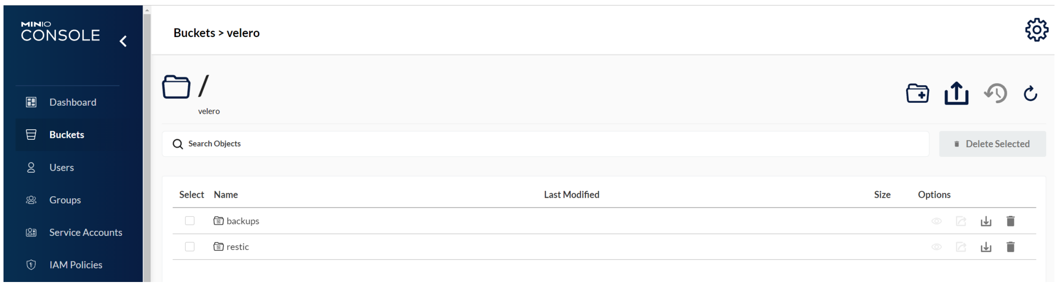
Step 3 Check the backup status.

```
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 restic path is the PV data backup path.



Select	Name	Last Modified	Size	Options
	backups			
	restic			

----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**.

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

3.1.6 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.

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:

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.

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.

```
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 the 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 3-4 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.

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.

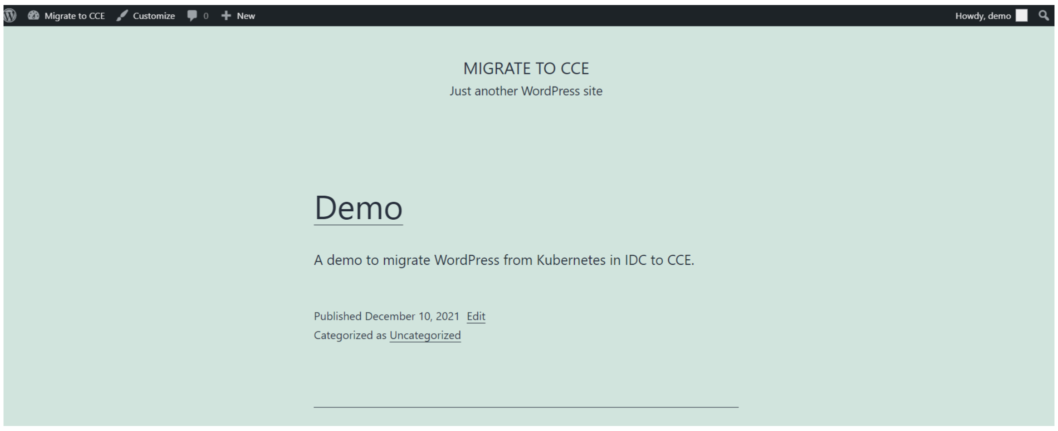
----End

3.1.7 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.



MIGRATE TO CCE
Just another WordPress site

Demo

A demo to migrate WordPress from Kubernetes in IDC to CCE.

Published December 10, 2021 Edit
Categorized as Uncategorized

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.

3.1.8 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-restic \
   --restic-pod-cpu-request 500m \
   --restic-pod-mem-request 1Gi \
   --restic-pod-cpu-limit 1000m \
   --restic-pod-mem-limit 1Gi
```

After Velero is installed:

Step 1 Edit the YAML files of the Velero and Restic workloads in the **velero** namespace.

```
kubectl edit deploy velero -n velero
kubectl edit deploy restic -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 Disaster Recovery

4.1 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 Security

5.1 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.

Disabling the Automatic Token Mounting Function of the Default Service Account

By default, Kubernetes associates the default service account with every pod. That is, 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. Set 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 the **automountServiceAccountToken** field 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 with service accounts for 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.

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.

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.

- 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.

5.2 Node Security

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**.

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.

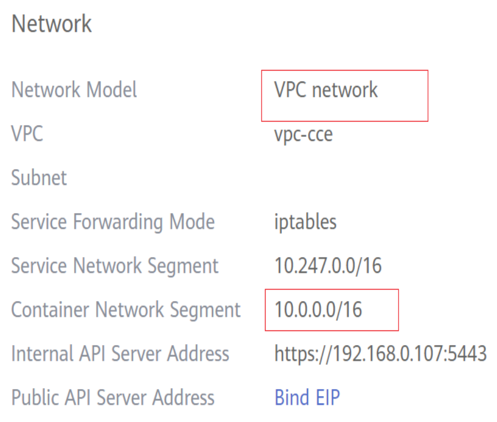
For details about how to restore the metadata, see the "Notes" section in "Obtaining Metadata".

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.


Network

Network Model	VPC network
VPC	vpc-cce
Subnet	
Service Forwarding Mode	iptables
Service Network Segment	10.247.0.0/16
Container Network Segment	10.0.0.0/16
Internal API Server Address	https://192.168.0.107:5443
Public API Server Address	Bind EIP

Step 2 Prevent the container from obtaining host metadata.

- VPC network
  a. Log in to each node in the CCE 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, you are advised to 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 CCE 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, you are advised to 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
  ```

----End

5.3 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.

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 Log in to each node in the CCE 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.

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

5.4 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 Auto Scaling

6.1 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.

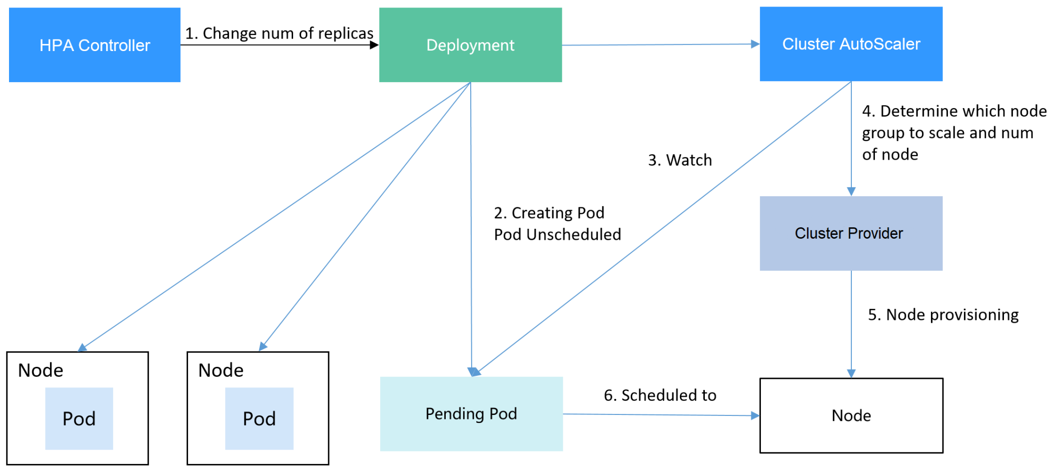
Solution

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 6-1**, 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.

Figure 6-1 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** is the image repository address, which 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 {*Image repository address***}/group/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 {*Image repository address***}/group/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 node pool parameters, add a node with 2 vCPUs and 4 GiB memory, and enable auto scaling.

- **Nodes**: Set it to **1**, indicating that one node is created by default when a node pool is created.
- **Auto Scaling**: Enable the option, meaning that nodes will be automatically created or deleted in the node pool based on the cluster loads.
- **Max. Nodes**: Set it to **5**, indicating the maximum number of nodes in a node pool.
- **Specifications**: 2 vCPUs | 4 GiB

Retain the defaults for other parameters.

Step 3 Click **Add-ons** on the left of the cluster console, click **Edit** under the autoscaler add-on, modify the add-on configuration, enable **Auto node scale-in**, and configure scale-in parameters. For example, trigger scale-in when the node resource utilization is less than 50%.

After the preceding configurations, scale-out is performed based on the pending status of the pod and scale-in is triggered when the node resource utilization decreases.

Step 4 Click **Node Scaling** on the left of the cluster console and click **Create Node Scaling Policy** in the upper right corner. Node scaling policies added here trigger scale-out based on the CPU/memory allocation rate or periodically.

As shown in the following figure, when the cluster CPU allocation rate is greater than 70%, one node will be added. 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.

----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.

```
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
```

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. .

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).

----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 Monitoring

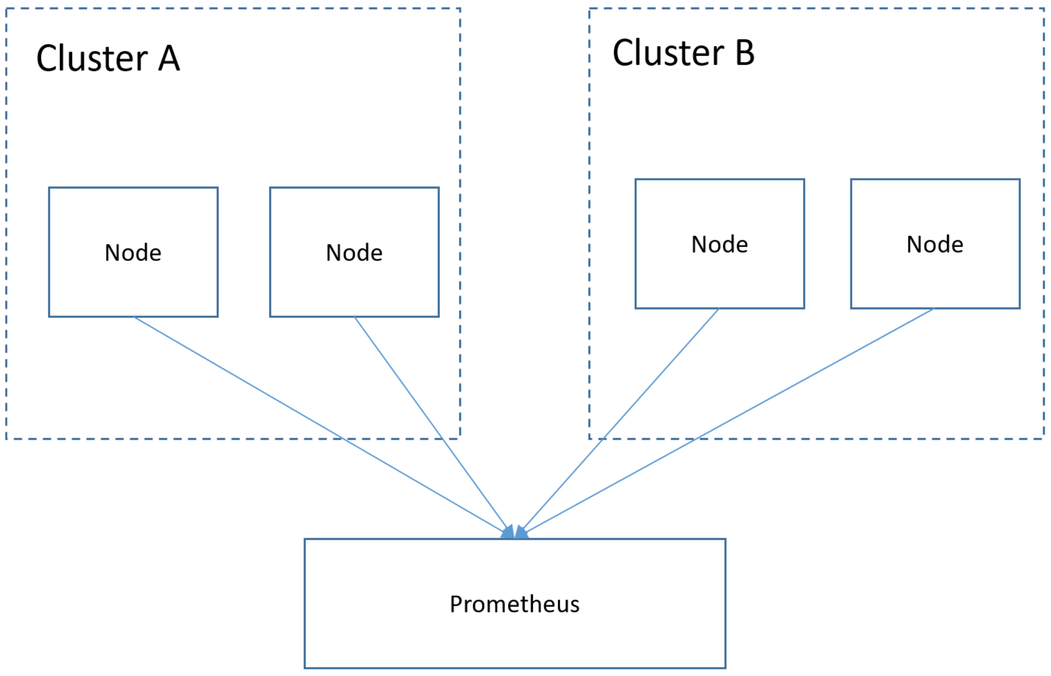
7.1 Using Prometheus for Multi-cluster Monitoring

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.

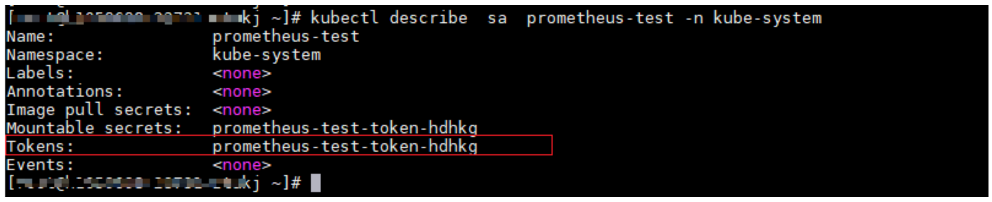
NOTE

- 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.
- 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.

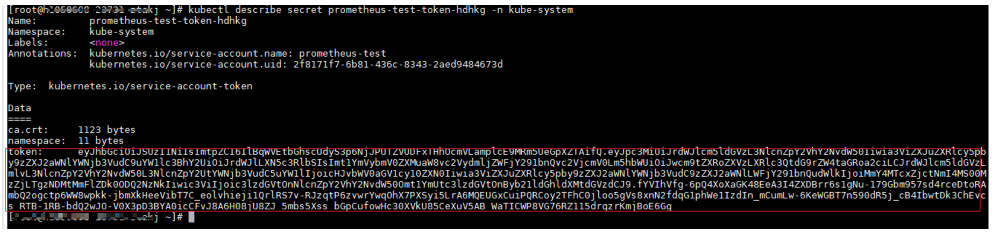
Obtain the **serviceaccount** information.

kubectl describe sa prometheus-test -n kube-system



```
[[redacted]kj ~]# kubectl describe  sa  prometheus-test -n kube-system
Name:                prometheus-test
Namespace:           kube-system
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   prometheus-test-token-hdhkg
Tokens:              prometheus-test-token-hdhkg
Events:              <none>
[[redacted]kj ~]#
```

kubectl describe secret prometheus-test-token-hdhkg -n kube-system

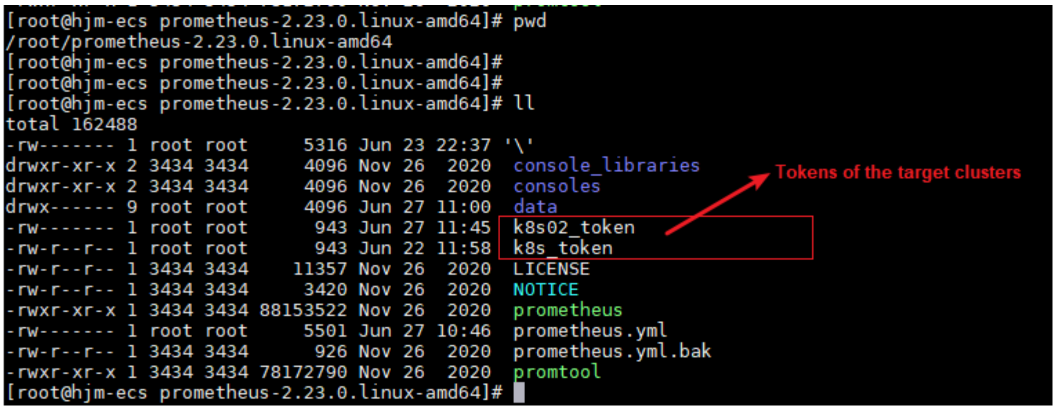


```
[root@h[redacted]kj ~]# kubectl describe secret prometheus-test-token-hdhkg -n kube-system
Name:         prometheus-test-token-hdhkg
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: prometheus-test
              kubernetes.io/service-account.uid: 2f8171f7-6b81-436c-8343-2aed9484673d

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1123 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IlBqWVEtbGhscUdyS3p6NjJPUTZVUDFxTHhOcmVLamplcE9MRm5UeGpXZTAifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pb
y9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiOiJwcm9tZXRoZXVzLXRlc3QtdG9rZW4taGRoa2ciLCJrdWJlcm5ldGVzL
mlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoicHJvbWV0aGV1cy10ZXN0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQudWlkIjoiMmY4MTcxZjctNmI4MS00M
zZjLTgzNDMtMmFlZDk0ODQ2NzNkIiwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOnByb21ldGhldXMtdGVzdCJ9.fYVIhVfg-6pQ4XoXaGK48EeA3I4ZXDBrr6s1gNu-179Gbm957sd4rceDtoRA
mbQ2ogctp6WW8wpkk-jbmXkHeeVibT7C_eolvhieji1QrlRS7v-RJzqtP6zvwrYwqOhX7PXSyi5LrA6MQEUGxCuiPQRCoy2TFhC0jloo5gVs8xnN2fdqG1phWe1IzdIn_mCumLw-6KeWGBT7n590dR5j_cB4IbwtDk3ChEvc
s RTB-1RB-bdQ2wJO-V0X3pD3BYA0icCFvJ8A6H08jU8ZJ 5mbs5Xss bGpCufowHc30XVkU85CeXuV5AB WaTICWP8VG76RZ115drqzrKmjBoE6Gg
[[redacted]kj ~]#
```

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.



```
[root@hjm-ecs prometheus-2.23.0.linux-amd64]# pwd
/root/prometheus-2.23.0.linux-amd64
[root@hjm-ecs prometheus-2.23.0.linux-amd64]#
[root@hjm-ecs prometheus-2.23.0.linux-amd64]#
[root@hjm-ecs prometheus-2.23.0.linux-amd64]# ll
total 162488
-rw------- 1 root root     5316 Jun 23 22:37 '\'
drwxr-xr-x 2 3434 3434     4096 Nov 26  2020 console_libraries
drwxr-xr-x 2 3434 3434     4096 Nov 26  2020 consoles
drwx------ 9 root root     4096 Jun 27 11:00 data
-rw------- 1 root root      943 Jun 27 11:45 k8s02_token
-rw-r--r-- 1 root root      943 Jun 22 11:58 k8s_token
-rw-r--r-- 1 3434 3434    11357 Nov 26  2020 LICENSE
-rw-r--r-- 1 3434 3434     3420 Nov 26  2020 NOTICE
-rwxr-xr-x 1 3434 3434 88153522 Nov 26  2020 prometheus
-rw------- 1 root root     5501 Jun 27 10:46 prometheus.yml
-rw-r--r-- 1 3434 3434      926 Nov 26  2020 prometheus.yml.bak
-rwxr-xr-x 1 3434 3434 78172790 Nov 26  2020 promtool
[root@hjm-ecs prometheus-2.23.0.linux-amd64]#
```

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.

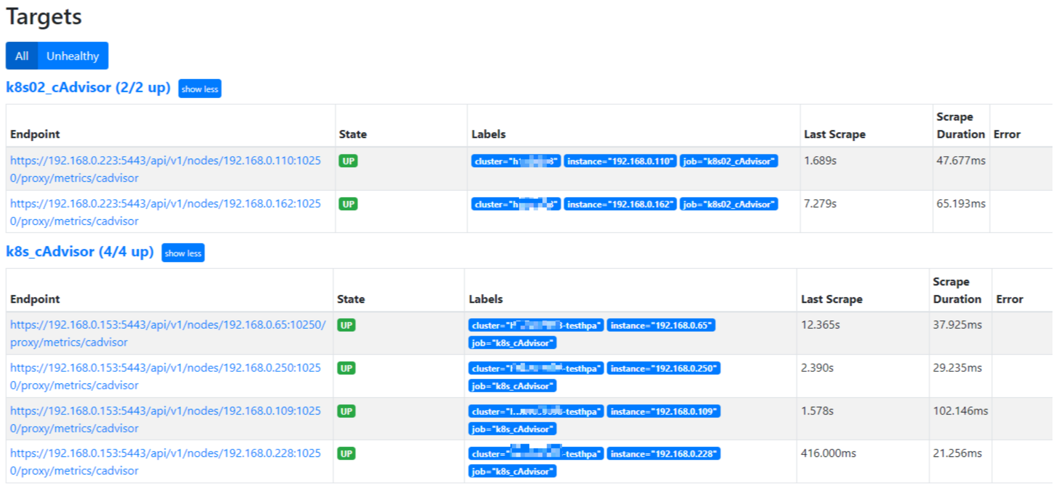
```
  - job_name: k8s_cAdvisor
    scheme: https
    bearer_token_file: k8s_token # Token file in the previous step.
    tls_config:
      insecure_skip_verify: true
    kubernetes_sd_configs:  # kubernetes automatic discovery configuration
    - role: node    # Automatic discovery of the node type
      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.



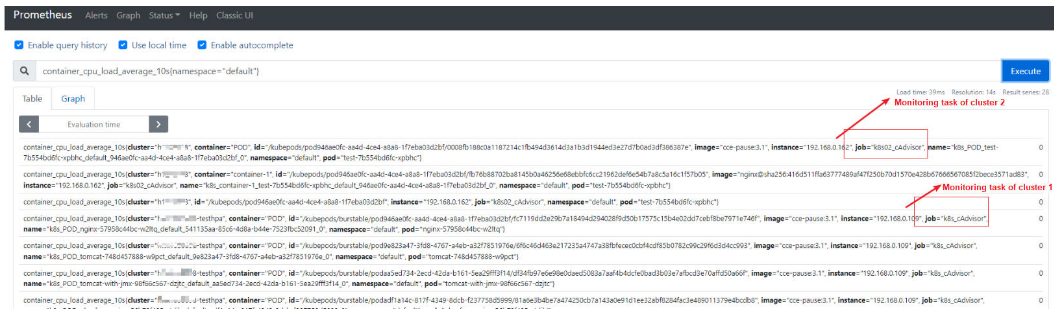
Targets

k8s02_cAdvisor (2/2 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://192.168.0.223:5443/api/v1/nodes/192.168.0.110:10250/proxy/metrics/cadvisor	UP	cluster="h[redacted]8" instance="192.168.0.110" job="k8s02_cAdvisor"	1.689s	47.677ms	
https://192.168.0.223:5443/api/v1/nodes/192.168.0.162:10250/proxy/metrics/cadvisor	UP	cluster="h[redacted]8" instance="192.168.0.162" job="k8s02_cAdvisor"	7.279s	65.193ms	

k8s_cAdvisor (4/4 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://192.168.0.153:5443/api/v1/nodes/192.168.0.65:10250/proxy/metrics/cadvisor	UP	cluster="[redacted]-testhpa" instance="192.168.0.65" job="k8s_cAdvisor"	12.365s	37.925ms	
https://192.168.0.153:5443/api/v1/nodes/192.168.0.250:10250/proxy/metrics/cadvisor	UP	cluster="[redacted]-testhpa" instance="192.168.0.250" job="k8s_cAdvisor"	2.390s	29.235ms	
https://192.168.0.153:5443/api/v1/nodes/192.168.0.109:10250/proxy/metrics/cadvisor	UP	cluster="[redacted]-testhpa" instance="192.168.0.109" job="k8s_cAdvisor"	1.578s	102.146ms	
https://192.168.0.153:5443/api/v1/nodes/192.168.0.228:10250/proxy/metrics/cadvisor	UP	cluster="[redacted]-testhpa" instance="192.168.0.228" job="k8s_cAdvisor"	416.000ms	21.256ms	



Enable query history ✓ Use local time ✓ Enable autocomplete ✓

```
container_cpu_load_average_10s{namespace="default"}
```

Load time: 39ms   Resolution: 14s   Result series: 28

```
container_cpu_load_average_10s{cluster="h[redacted]8", container="POD", id="/kubepods/pod946ae0fc-aa4d-4ce4-a8a8-1f7eba03d2bf/0008fb188c0a1187214c1fb494d3614d3a1b3d1944ed3e27d7b0ad3df386387e", image="cce-pause:3.1", instance="192.168.0.162", job="k8s02_cAdvisor", name="k8s_POD_test-
7b554bd6fc-xpbhc_default_946ae0fc-aa4d-4ce4-a8a8-1f7eba03d2bf_0", namespace="default", pod="test-7b554bd6fc-xpbhc"}    0
container_cpu_load_average_10s{cluster="h[redacted]8", container="container-1", id="/kubepods/pod946ae0fc-aa4d-4ce4-a8a8-1f7eba03d2bf/fb76b88702ba8145b0a46256e68ebbfc6cc21962def6e54b7a8c5a16c1f57b05", image="nginx@sha256:416d511ffa63777489af47f250b70d1570e428b67666567085f2bece3571ad83",    0
instance="192.168.0.162", job="k8s02_cAdvisor", name="k8s_container-1_test-7b554bd6fc-xpbhc_default_946ae0fc-aa4d-4ce4-a8a8-1f7eba03d2bf_0", namespace="default", pod="test-7b554bd6fc-xpbhc"}
container_cpu_load_average_10s{cluster="h[redacted]8", id="/kubepods/pod946ae0fc-aa4d-4ce4-a8a8-1f7eba03d2bf", instance="192.168.0.162", job="k8s02_cAdvisor", namespace="default", pod="test-7b554bd6fc-xpbhc"}    0
container_cpu_load_average_10s{cluster="[redacted]-testhpa", container="POD", id="/kubepods/burstable/pod946ae0fc-aa4d-4ce4-a8a8-1f7eba03d2bf/fc7119dd2e29b7a18494d294028f9d50b17575c15b4e02dd7cebf8be7971e746f", image="cce-pause:3.1", instance="192.168.0.109", job="k8s_cAdvisor",    0
name="k8s_POD_nginx-57958c44bc-w2ltq_default_541135aa-85c6-4d8a-b44e-7523fbc52091_0", namespace="default", pod="nginx-57958c44bc-w2ltq"}
container_cpu_load_average_10s{cluster="[redacted]-testhpa", container="POD", id="/kubepods/burstable/pod9e823a47-3fd8-4767-a4eb-a32f7851976e/6f6c46d463e217235a4747a38fbfecec0cbf4cdf85b0782c99c29f6d3d4cc993", image="cce-pause:3.1", instance="192.168.0.109", job="k8s_cAdvisor",    0
name="k8s_POD_tomcat-748d457888-w9pct_default_9e823a47-3fd8-4767-a4eb-a32f7851976e_0", namespace="default", pod="tomcat-748d457888-w9pct"}
container_cpu_load_average_10s{cluster="[redacted]8-testhpa", container="POD", id="/kubepods/burstable/podaa5ed734-2ecd-42da-b161-5ea29fff3f14/df34fb97e6e98e0daed5083a7aaf4b4dcfe0bad3b03e7afbcd3e70affd50a66f", image="cce-pause:3.1", instance="192.168.0.109", job="k8s_cAdvisor",    0
name="k8s_POD_tomcat-with-jmx-98f66c567-dzjtc_default_aa5ed734-2ecd-42da-b161-5ea29fff3f14_0", namespace="default", pod="tomcat-with-jmx-98f66c567-dzjtc"}
container_cpu_load_average_10s{cluster="[redacted]-testhpa", container="POD", id="/kubepods/burstable/podadf1a14c-817f-4349-8dcb-f237758d5999/81a6e3b4be7a474250cb7a143a0e91d1ee32abf8284fac3e489011379e4bcdb8", image="cce-pause:3.1", instance="192.168.0.109", job="k8s_cAdvisor",    0
```

----End

8 Cluster

8.1 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.

  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.
- 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.

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 the temporary server /tmp/example.package.

Step 2 Install Packer.

Download and install the **HashiCorp Packer**. For details, see **Install Packer**.

NOTE

Install Packer of v1.7.2 or later.

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

# 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 = "= 0.4.0"
        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
```

Table 8-1 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	Submit a service ticket to obtain the image ID.

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.

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
```

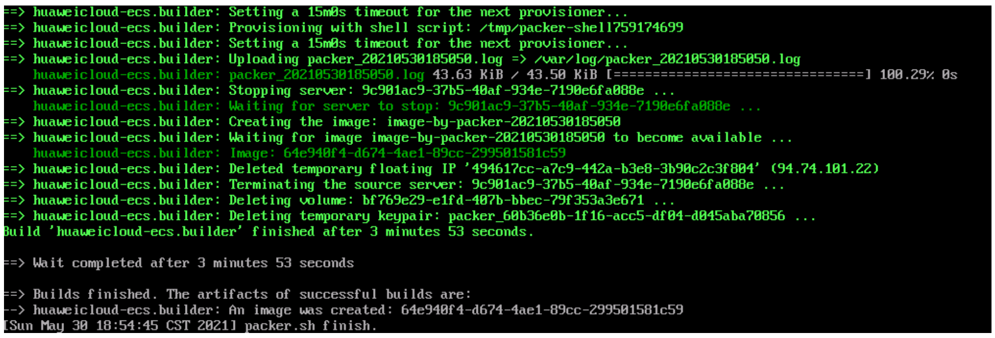
NOTE

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-shell759174699
==> 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

8.2 Connecting to Multiple Clusters Using kubectl

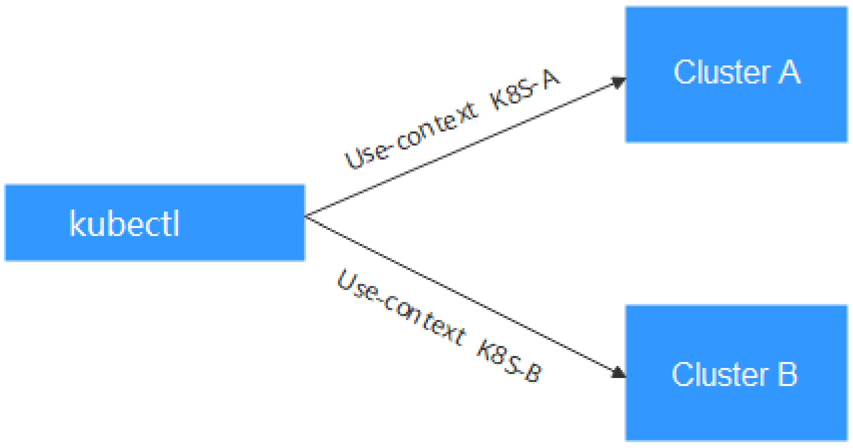
Background

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 8-1 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 commands to copy the 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'.
```

8.3 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. 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 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
sda                   8:0    0   50G  0 disk
└─sda1                8:1    0   50G  0 part /
sdb                   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/sdb
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
sda                                   8:0    0   50G  0 disk
└─sda1                                8:1    0   50G  0 part /
sdb                                   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                    # Thin pool space.
│   ...
├─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/sdb
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/sdb
lvextend -l+100%FREE -n vgpaas/dockersys
resize2fs /dev/vgpaas/dockersys
```

----End

9 Networking

9.1 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

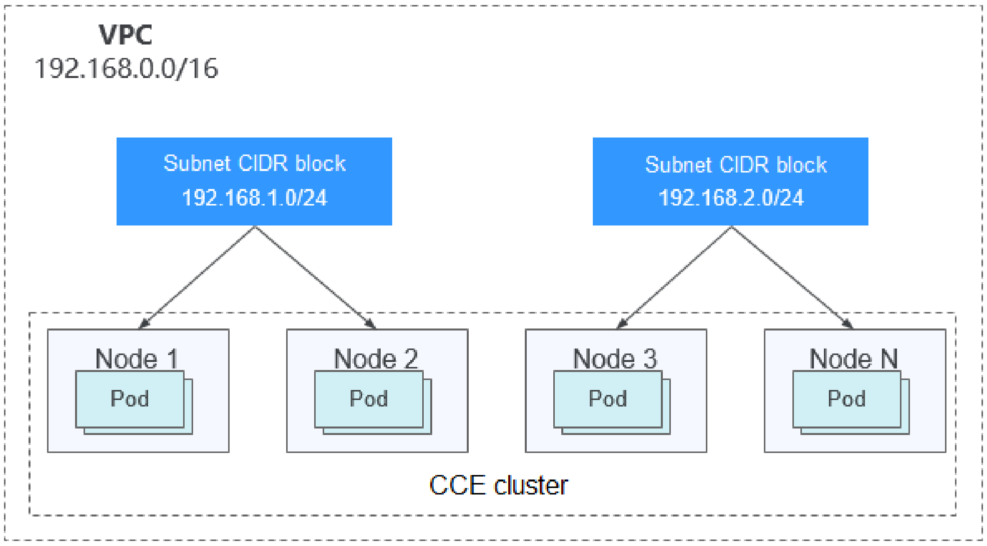
- **VPC CIDR Block**

  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 9-1 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.

- **Service CIDR Block**

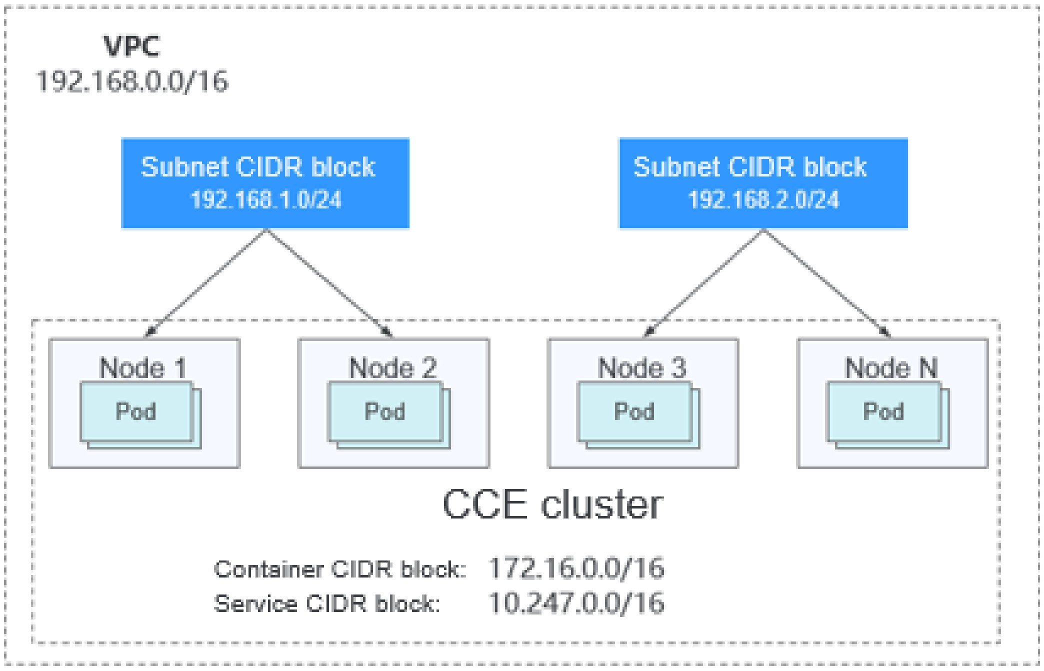
  Service is also a Kubernetes concept. Each Service has an address. When creating a cluster on CCE, you can specify the Service CIDR block. Similarly, the Service CIDR block cannot overlap with the subnet CIDR block or the container CIDR block. The Service CIDR block can be used only within a cluster.

Single-VPC Single-Cluster Scenarios

CCE Clusters: include clusters in VPC network model and container tunnel network model. **Figure 9-2** 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 9-2 Network CIDR block planning in single-VPC single-cluster scenarios (CCE cluster)



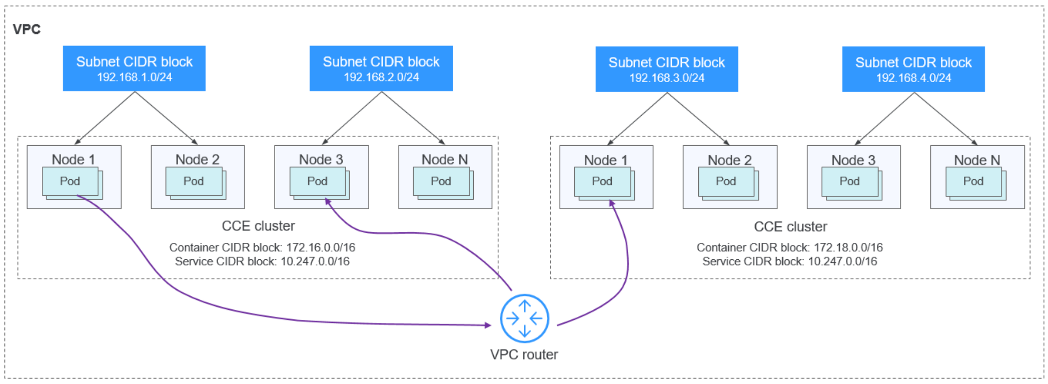
Single-VPC Multi-Cluster Scenarios

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 9-3** 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, CCE clusters are partially interconnected. A pod of a cluster can directly access the pods of another cluster, but cannot access the Services of the cluster.
- 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 9-3 VPC network - multi-cluster scenario

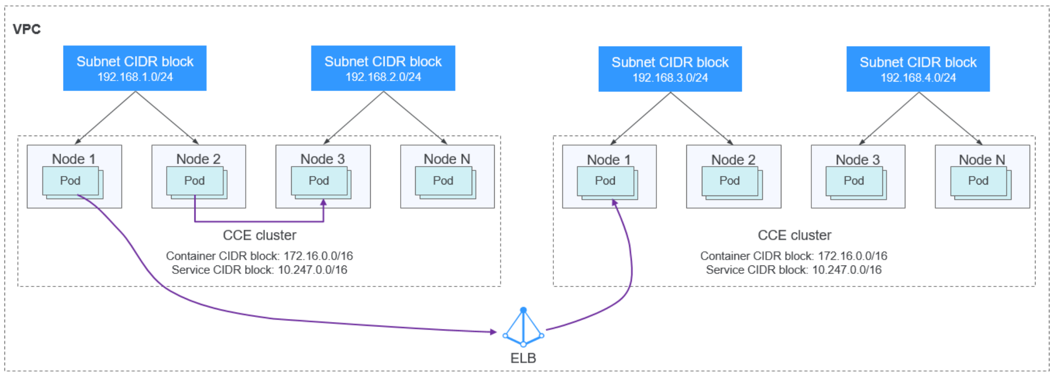

VPC
Subnet CIDR block 192.168.1.0/24
Subnet CIDR block 192.168.2.0/24
Subnet CIDR block 192.168.3.0/24
Subnet CIDR block 192.168.4.0/24
Node 1
Node 2
Node 3
Node N
Pod
CCE cluster
Container CIDR block: 172.16.0.0/16
Service CIDR block: 10.247.0.0/16
CCE cluster
Container CIDR block: 172.18.0.0/16
Service CIDR block: 10.247.0.0/16
VPC router

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 9-4** 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: The container CIDR blocks of all clusters can overlap. In this case, pods in different clusters cannot be directly accessed using IP addresses. It is recommended that ELB be used for the cross-cluster access between containers.
- 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 9-4 Tunnel network - multi-cluster scenario


VPC
Subnet CIDR block 192.168.1.0/24
Subnet CIDR block 192.168.2.0/24
Subnet CIDR block 192.168.3.0/24
Subnet CIDR block 192.168.4.0/24
Node 1
Node 2
Node 3
Node N
Pod
CCE cluster
Container CIDR block: 172.16.0.0/16
Service CIDR block: 10.247.0.0/16
CCE cluster
Container CIDR block: 172.16.0.0/16
Service CIDR block: 10.247.0.0/16
ELB

Coexistence of Clusters in Multi-Network

When a VPC contains clusters created with different network models, comply with the following rules when creating a cluster:

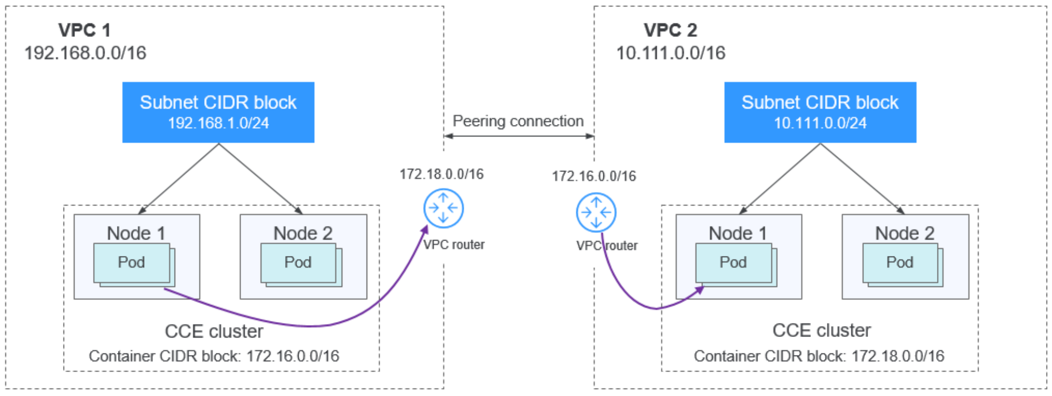
- VPC CIDR Block: In this scenario, all clusters are located in the same VPC CIDR block. Ensure that there are sufficient available IP addresses in the VPC.
- Subnet CIDR Block: Ensure that the subnet CIDR block does not overlap with the container CIDR block.
- Container CIDR Block: Ensure that the container 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

When two VPC networks are interconnected, you can configure the packets to be sent to the peer VPC in the route table.

In the VPC network model, after creating a peering connection, add routes for the peering connection to enable communication between the two VPCs.

Figure 9-5 VPC network - VPC interconnection scenario

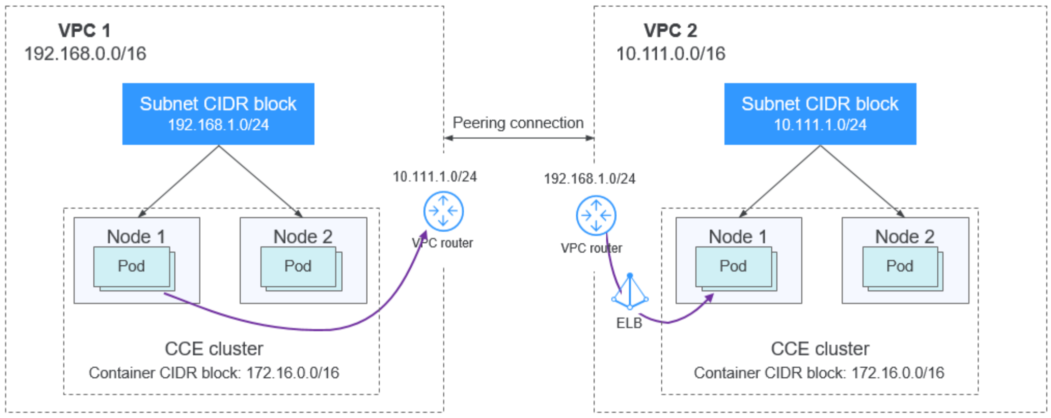


When creating a VPC peering connection between containers across VPCs, pay attention to the following points:

- 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.
- Add not only the peer VPC CIDR block but also the peer container CIDR block to the VPC routing tables at both ends. Note that this operation must be performed in the VPC route tables of the clusters.

In the Tunnel network model, after creating a peering connection, add routes for the peering connection to enable communication between the two VPCs.

Figure 9-6 Tunnel network - VPC interconnection scenario



Pay attention to the following:

- The VPCs of the clusters must not overlap.
- The container CIDR blocks of all clusters can overlap, so do the Service CIDR blocks.
- Add the peer subnet CIDR block to the route table of the VPC peering connection.

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.

9.2 Selecting a Network Model

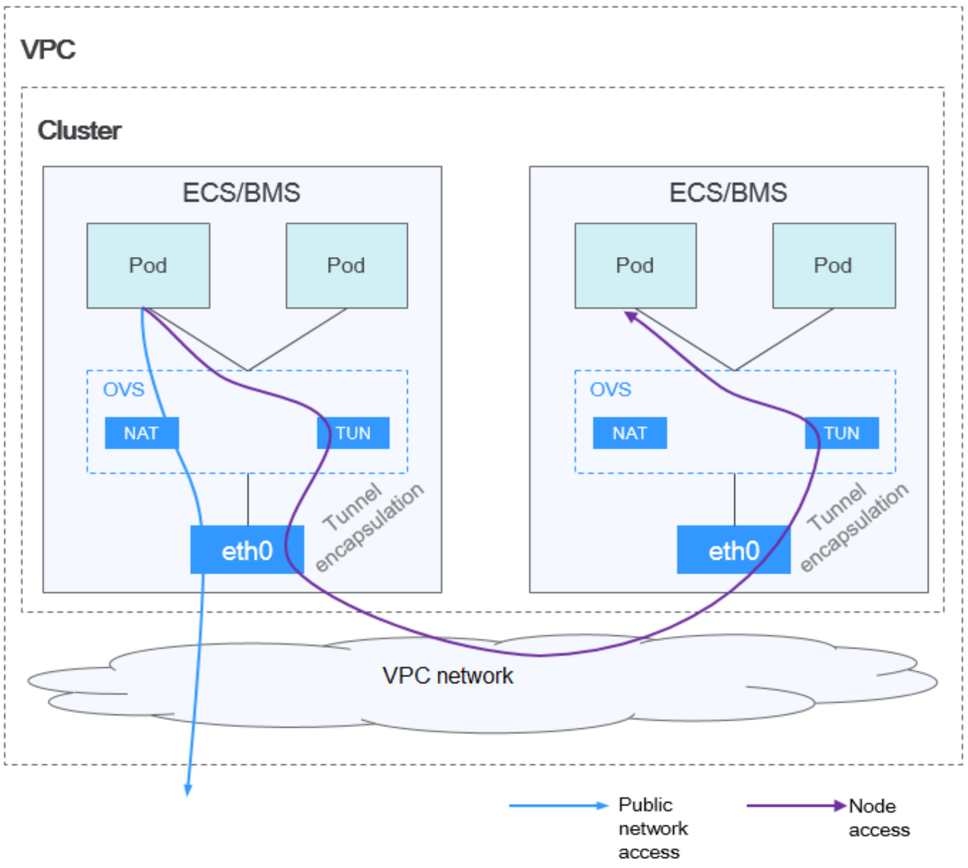
CCE uses proprietary, high-performance container networking add-ons to support the tunnel network and VPC network models.

CAUTION

After a cluster is created, the network model cannot be changed. Exercise caution when selecting a network model.

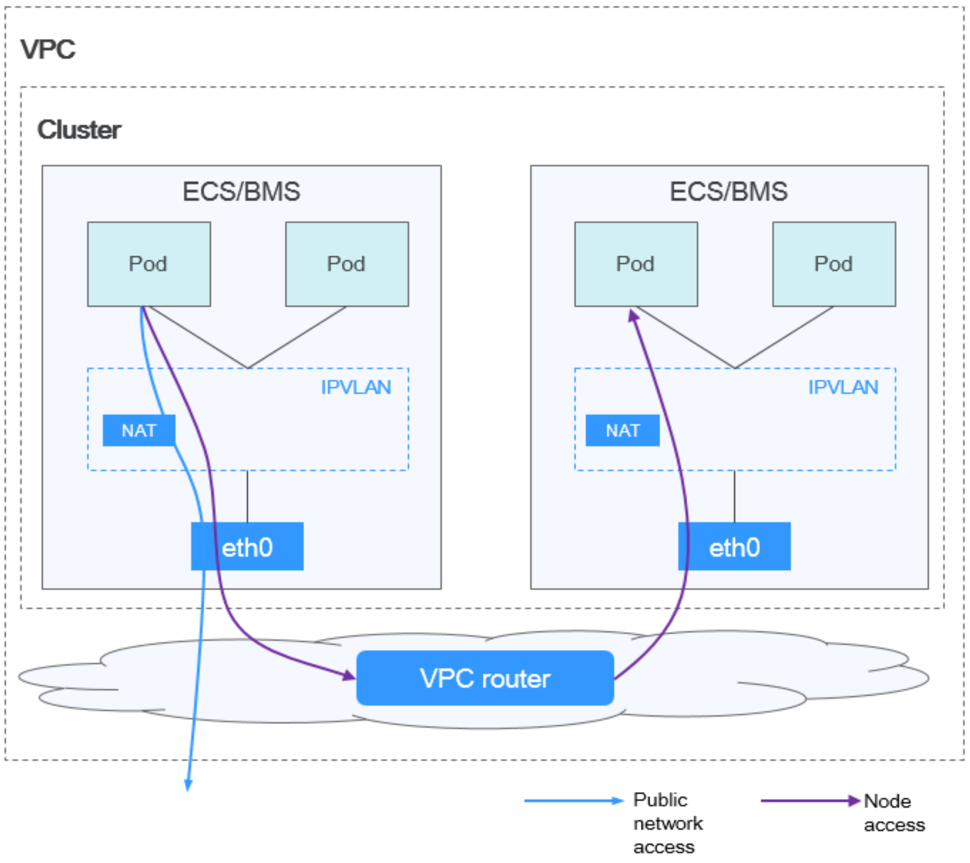
- **Tunnel network**: The container network is an overlay tunnel network on top of a VPC network and uses the VXLAN technology. This network model is applicable when there is no high requirements on performance. VXLAN encapsulates Ethernet packets as UDP packets for tunnel transmission. Though at some 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 9-7 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 9-8 VPC network



The following table lists the differences between the network models.

Table 9-1 Networking model comparison

Dimension	Tunnel Network	VPC Network
Core technology	OVS	IPvlan and VPC route
Applicable Clusters	CCE cluster	CCE cluster
Network isolation	Kubernetes native NetworkPolicy for pods	No
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).

Dimension	Tunnel Network	VPC 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.
Networking scale	A maximum of 2,000 nodes are supported.	By default, 200 nodes are supported. Each time a node is added to the cluster, a route is added to the VPC routing table. Therefore, the cluster scale is limited by the VPC route table.
Application Scenarios	• Common container services • Scenarios that do not have high requirements on network latency and bandwidth	• Scenarios that have high requirements on network latency and bandwidth • Containers communicate with VMs using a microservice registration framework, such as Dubbo and CSE.

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.

9.3 Implementing Sticky Session Through Load Balancing

Concepts

Session persistence is one of the most common while complex problems in load balancing.

Session persistence is also called sticky sessions. After the sticky session function is enabled, requests from the same client are distributed to the same backend ECS by the load balancer for better continuity.

In load balancing and sticky session, connection and session are two key concepts. When only load balancing is concerned, session and connection refer to the same thing.

Simply put, if a user needs to log in, it can be regarded as a session; otherwise, a connection.

The sticky session mechanism fundamentally conflicts with the basic functions of load balancing. A load balancer forwards requests from clients to multiple backend servers to avoid overload on a single server. However, sticky session requires that some requests be forwarded to the same server for processing. Therefore, select a proper sticky session mechanism based on the application environment.

Layer-4 Load Balancing (Service)

In layer-4 load balancing, source IP address-based sticky session (Hash routing based on the client IP address) can be enabled. To enable source IP address-based sticky session on Services, the following conditions must be met:

CCE clusters

1. **Service Affinity** of the Service is set to **Node level** (that is, the value of the **externalTrafficPolicy** field of the Service is **Local**).

   NOTE

   You do not need to set this parameter for CCE Turbo clusters.

2. Enable the source IP address-based sticky session in the load balancing configuration of the Service.
```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: 56dcc1b4-8810-480c-940a-a44f7736f0dc
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local   # You do not need to set this parameter for CCE Turbo clusters.
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```
3. Anti-affinity is enabled for the backend application corresponding to the Service.

Layer-7 Load Balancing (Ingress)

In layer-7 load balancing, sticky session based on HTTP cookies and app cookies can be enabled. To enable such sticky session, the following conditions must be met:

1. The application (workload) corresponding to the ingress is enabled with workload anti-affinity.
2. Node affinity is enabled for the Service corresponding to the ingress.

Procedure

Step 1 Create an Nginx workload.

Set the number of pods to 3 and set the 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 Creating a NodePort Service

Configure the sticky session in a Service. An ingress can connect to 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
    kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE      # HTTP cookie type.
    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            # Node port number.
  type: NodePort
  externalTrafficPolicy: Local   # Node-level forwarding.
```

You can also select **APP_COOKIE**.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN
    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 a Service.

```
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.autocreate:
      '{
          "type":"public",
          "bandwidth_name":"cce-bandwidth-test",
          "bandwidth_chargemode":"traffic",
          "bandwidth_size":1,
          "bandwidth_sharetype":"PER",
          "eip_type":"5_bgp"
        }'
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, access the load balancer details page, and check whether the sticky session feature is enabled.

----End

9.4 Pre-Binding Container ENI for CCE Turbo Clusters

In the Cloud Native Network 2.0 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 1.23.5-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 pools is not supported.
- 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.

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 9-2 Parameters of the dynamic ENI pre-binding policy

Parameter	Default Value	Description	Suggestion
nic-minimum-target	10	Minimum number of ENIs bound to a node. The value can be a number or a percentage. ● Value: The value must be a positive integer. For example, 10 indicates that at least 10 ENIs are bound to a node. If the ENI quota of a node is exceeded, the ENI quota is used. ● Percentage: The value ranges from 1% to 100%. For example, 10%. If the ENI quota of a node is 128, at least 12 (rounded down) ENIs are bound to the node. Set both **nic-minimum-target** and **nic-maximum-target** to the same value or percentage.	Set these parameters based on the number of pods.

Parameter	Default Value	Description	Suggestion
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 NICs. 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 value can be a number or a percentage. ● Value: The value must be a positive integer. For example, 0. The check on the maximum number of the pre-bound ENIs is disabled. If the ENI quota of a node is exceeded, the ENI quota is used. ● Percentage: The value ranges from 1% to 100%. For example, 50%. If the ENI quota of a node is 128, the maximum number of the pre-bound ENI is 64 (rounded down). Set both **nic-minimum-target** and **nic-maximum-target** to the same value or percentage.	Set these parameters based on the number of pods.
nic-warm-target	2	Extra ENIs will be pre-bound after the **nic-minimum-target** is used up in a pod. The value can only 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 or 16% ● nic-maximum-target: 60 or 47% ● 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 or 40% ● nic-maximum-target: 120 or 50% ● 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.

Step 3 In the **Manage Component** window that is displayed on the sidebar, select **Networking Components**. For details about the parameter values, 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** on the left, and click the **Node Pools** tab on the right.

Step 3 Choose **More** > **Manage** next to the node pool name.

Step 4 In the **Manage Component** window that is displayed on the sidebar, select **Networking Components**. For details about the parameter values, 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 Storage

10.1 Expanding the Storage Space

The storage classes that can be expanded for CCE nodes are as follows:

Table 10-1 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**
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
sda                   8:0    0   50G  0 disk
└─sda1                8:1    0   50G  0 part /
sdb                   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/sdb
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
sda                                   8:0    0   50G  0 disk
└─sda1                                8:1    0   50G  0 part /
sdb                                   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                   # Thin pool space.
│   ...
├─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/sdb
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/sdb
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 the following commands on the node to add the new disk capacity to the Kubernetes disk:

```
pvresize /dev/sdb
lvextend -l+100%FREE -n vgpaas/kubernetes
resize2fs /dev/vgpaas/kubernetes
```

----End

Expanding the Capacity of a Data Disk Used by Pod (basesize)

Step 1 Log in to the CCE console and click the name of the target cluster in the cluster list.

Step 2 Choose **Nodes** from the navigation pane.

Step 3 Select the target node and choose **More** > **Reset Node** in the **Operation** column.

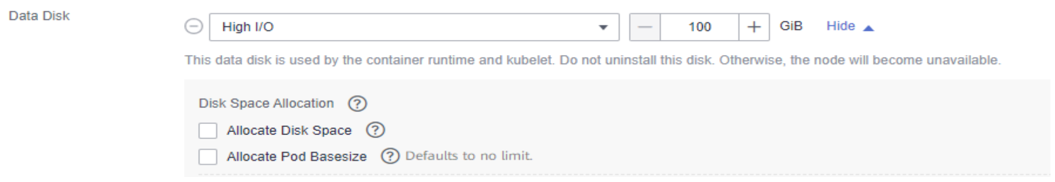
NOTICE

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:


Data Disk
High I/O 100 GiB Hide
This data disk is used by the container runtime and kubelet. Do not uninstall this disk. Otherwise, the node will become unavailable.
Disk Space Allocation
☐ Allocate Disk Space
☐ Allocate Pod Basesize Defaults to no limit.

Storage Settings: Click **Expand** next to the data disk to set the following parameters:

- **Allocate Disk Space**: storage space used by the container engine to store the Docker/containerd working directory, container image data, and image metadata. Defaults to 90% of the data disk.
- **Allocate Pod Basesize**: CCE allows you to set an upper limit for the disk space occupied by each workload pod (including the space occupied by container images). This setting prevents the pods from taking all the disk space available, which may cause service exceptions. It is recommended that the value be smaller than or equal to 80% of the container engine space.

NOTE

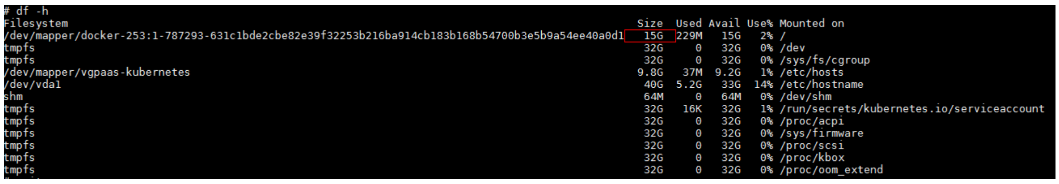
- The capability of customizing pod basesize is related to the node OS and container storage rootfs.
  - When the rootfs uses Device Mapper, the node supports custom pod basesize. The default storage space of a single container is 10 GiB.
  - When the rootfs uses OverlayFS, most nodes do not support custom pod basesize. The storage space of a single container is not limited and defaults to the container engine space.

    Only EulerOS 2.9 nodes in clusters of 1.19.16, 1.21.3, 1.23.3, and later versions support custom pod basesize.
- In the case of using Docker on EulerOS 2.9 nodes, **basesize** will not take effect if **CAP_SYS_RESOURCE** or **privileged** is configured for a container.

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** from the navigation pane, and click the **PersistentVolumeClaims (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 SFS Turbo, expand the capacity on the SFS console and then change the capacity in the PVC.

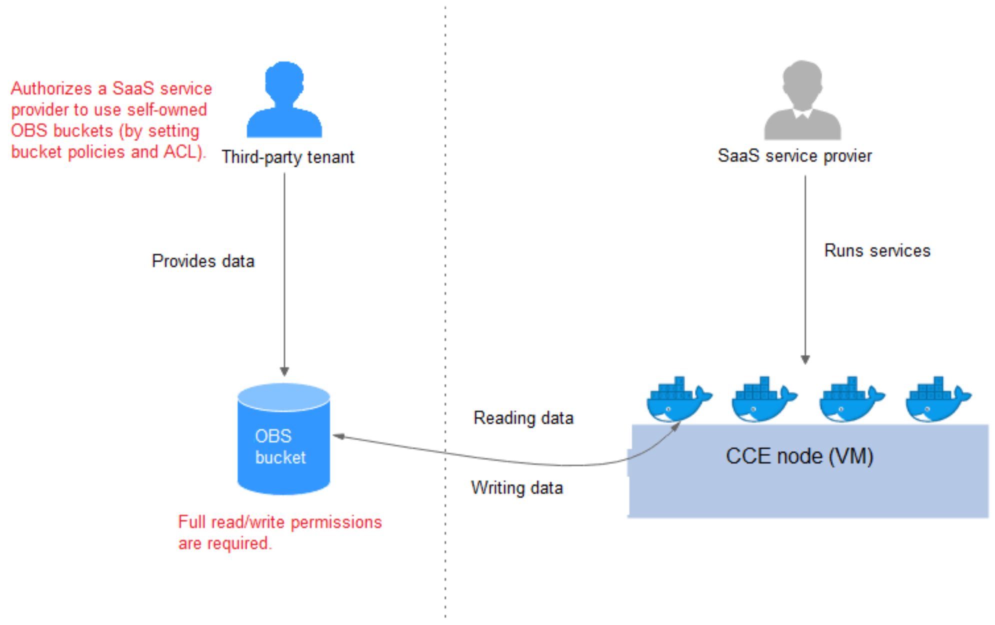
10.2 Mounting an Object Storage Bucket of a Third-Party Tenant

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 10-1**.

Figure 10-1 Mounting an OBS bucket of a third-party tenant


Authorizes a SaaS service provider to use self-owned OBS buckets (by setting bucket policies and ACL).
Third-party tenant
SaaS service provier
Provides data
Runs services
OBS bucket
Reading data
Writing data
CCE node (VM)
Full read/write permissions are required.

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

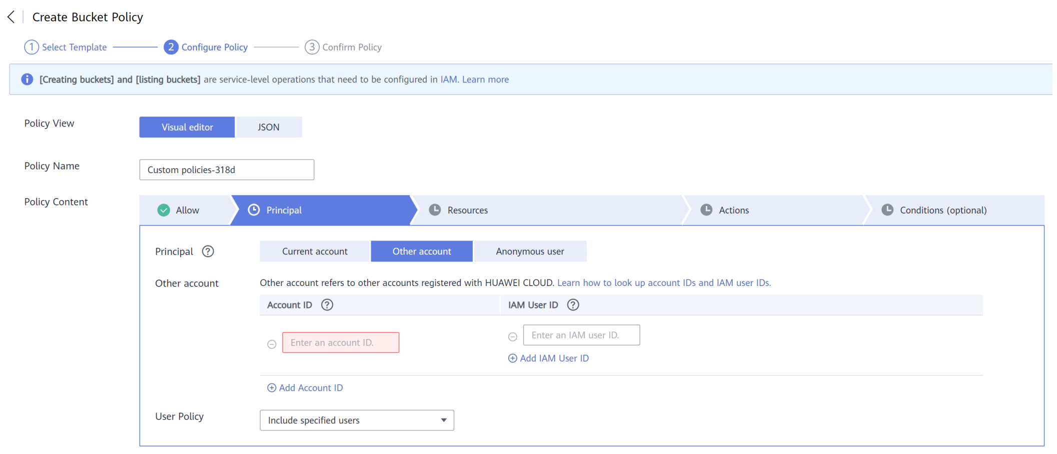
The following uses an OBS bucket as an example to describe how to set a bucket policy and bucket ACL to authorize the SaaS service provider. The configuration for an OBS parallel file system is the same.

Step 1 Log in to the OBS console.

Step 2 In the bucket list, click a bucket name to access the **Overview** page.

Step 3 In the navigation pane, choose **Permissions** > **Bucket Policy**. In the right pane, click **Create**. In the Custom policy area, click **Create Custom Policy**.

Figure 10-2 Creating a bucket policy



Create Bucket Policy

1 Select Template — 2 Configure Policy — 3 Confirm Policy

[Creating buckets] and [listing buckets] are service-level operations that need to be configured in IAM. Learn more

Policy View: Visual editor | JSON

Policy Name: Custom policies-318d

Policy Content: Allow > Principal > Resources > Actions > Conditions (optional)

Principal: Current account | Other account | Anonymous user

Other account: Other account refers to other accounts registered with HUAWEI CLOUD. Learn how to look up account IDs and IAM user IDs.

Account ID	IAM User ID
Enter an account ID.	Enter an IAM user ID.
	Add IAM User ID

Add Account ID

User Policy: Include specified users

- **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:        #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**.
- **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:
      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**.
   – **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.

10.3 Dynamically Creating and Mounting Subdirectories of 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.
- Nodes running Ubuntu OS do not support this function if they use the Docker container engine.

Creating an SFS Turbo Volume of the subpath Type

CAUTION

Do not expand, disassociate, or delete a **subpath** volume.

Step 1 Import an SFS Turbo file system that is located in the same VPC and subnet as the cluster.

Step 2 Create a StorageClass YAML file, for example, **sfsturbo-sc-test.yaml**.

The following is an example:

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  name: sfsturbo-sc-test
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 **Setting 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 configuration. 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 by the StorageClass are located in the 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 the recycling policy is set to **Delete**, the original PV file will be archived when the PVC is deleted. The archive directory is named in the format of *archived-$PV name.timestamp*. If this parameter is set to **false**, the SFS Turbo sub-directory corresponding to the PV will be deleted. The default value is **true**.

Step 3 Run the **kubectl create -f sfsturbo-sc-test.yaml** command to create a StorageClass.

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-sc-test
  volumeMode: Filesystem
```

In this example:

- **name**: indicates the name of the PVC.
- **storageClassName**: specifies the name of the StorageClass created in the previous step.
- **storage**: In the subpath mode, this parameter is invalid. The storage capacity is limited by the total capacity of the SFS Turbo file system. If the total capacity of the SFS Turbo file system is insufficient, expand the capacity on the SFS Turbo page in a timely manner.

Step 5 Run the **kubectl create -f sfs-turbo-test.yaml** command to create a PVC.

----End

NOTE

It is meaningless to conduct capacity expansion on an SFS Turbo volume created in the subpath mode. This operation does not expand the capacity of the SFS Turbo file system. Ensure that the total capacity of the SFS Turbo file system is not used up.

Creating a Deployment and Mounting an Existing Volume

Step 1 Create a Deployment YAML file named **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 Deployment.
- **image**: specifies the image used by the Deployment.
- **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 Run the **kubectl create -f deployment-test.yaml** command to create a Deployment.

----End

Creating a StatefulSet That Uses a Volume Dynamically Created in subpath Mode

Step 1 Create a StatefulSet YAML file named **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'
          env:
            - name: PAAS_APP_NAME
              value: deploy-sfs-nfs-rw-in
            - name: PAAS_NAMESPACE
              value: default
            - name: PAAS_PROJECT_ID
              value: 8190a2a1692c46f284585c56fc0e2fb9
          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-sc-test
  serviceName: wwww
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
  revisionHistoryLimit: 10
```

In this example:

- **name**: indicates the name of the StatefulSet.
- **image**: specifies the image used by the StatefulSet.
- **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**: indicates the name of the created StorageClass.

Step 2 Run the **kubectl create -f statefulset-test.yaml** command to create a StatefulSet.

----End

10.4 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 set 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).

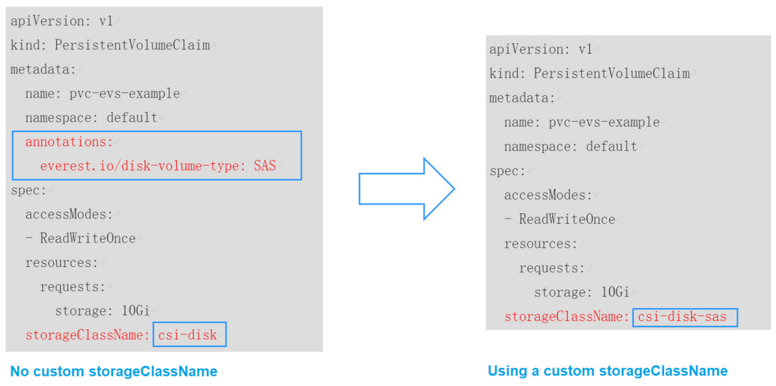
This configuration method may not work if you want to:

- 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.



```
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
```

No custom storageClassName

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: csi-disk-sas
```

Using a custom storageClassName

- 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
```

Default Storage Classes in CCE

Run the following command to query the supported storage classes.

```
# kubectl get sc
NAME                PROVISIONER                     AGE
csi-disk            everest-csi-provisioner         17d          # Storage class for EVS disks
csi-disk-topology   everest-csi-provisioner         17d          # Storage class for EVS disks with delayed 
association
csi-nas             everest-csi-provisioner         17d          # Storage class for SFS file systems
csi-obs             everest-csi-provisioner         17d          # Storage Class for OBS buckets
csi-sfsturbo        everest-csi-provisioner         17d          # Storage class for SFS Turbo file systems
```

Check the details of **csi-disk**. You can see that the type of the disk created by **csi-disk** is SAS by default.

```
# kubectl get sc csi-disk -oyaml
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  creationTimestamp: "2021-03-17T02:10:32Z"
  name: csi-disk
  resourceVersion: "760"
  selfLink: /apis/storage.k8s.io/v1/storageclasses/csi-disk
  uid: 4db97b6c-853b-443d-b0dc-41cdcb8140f2
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
```

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.

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 no 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 StorageClass

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.

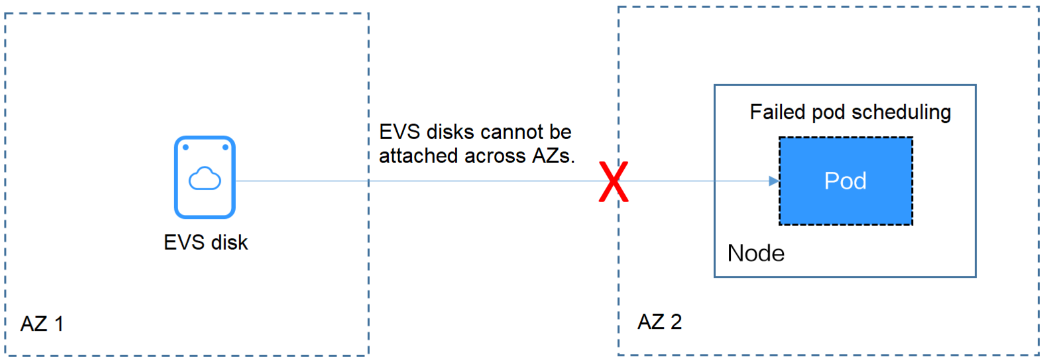
```
# kubectl create -f ssd-disk.yaml
persistentvolumeclaim/ssd-disk created
# kubectl get pvc
NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS   AGE
sas-disk   Bound    pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c   10Gi       RWO           csi-disk-sas
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.

10.5 Enabling Automatic Topology for EVS Disks When Nodes Are Deployed in Different AZs (csi-disk-topology)

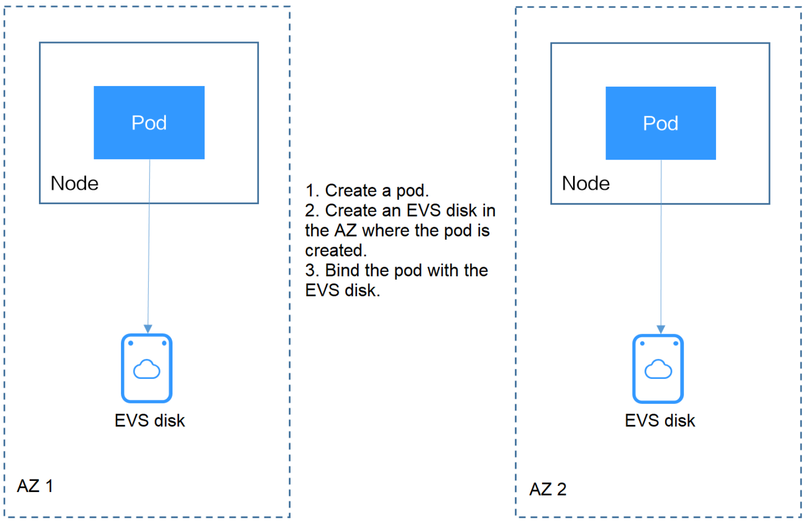
Background

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.


Pod
Node
1. Create a pod.
2. Create an EVS disk in the AZ where the pod is created.
3. Bind the pod with the EVS disk.
Pod
Node
EVS disk
EVS disk
AZ 1
AZ 2

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
```

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>
```

11 Container

11.1 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
```

11.2 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

11.3 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

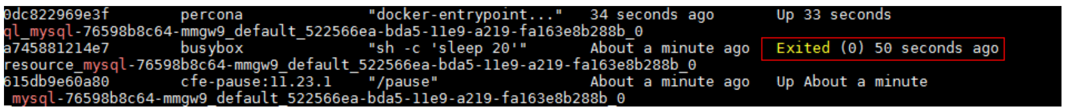
Information similar to the following is displayed:

```
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:11.23.1   "/pause"                 About a minute ago   Up About a minute
 mysql-76598b8c64-mmgw9_default_522566ea-bda5-11e9-a219-fa163e8b288b_0
```

----End

11.4 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 11-1 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
spec	Yes	Detailed description of the pod. For details, see **Table 11-2**.

Table 11-2 spec field description

Parameter	Mandatory	Description
hostAliases	Yes	Host alias
containers	Yes	For details, see **Table 11-3**.

Table 11-3 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

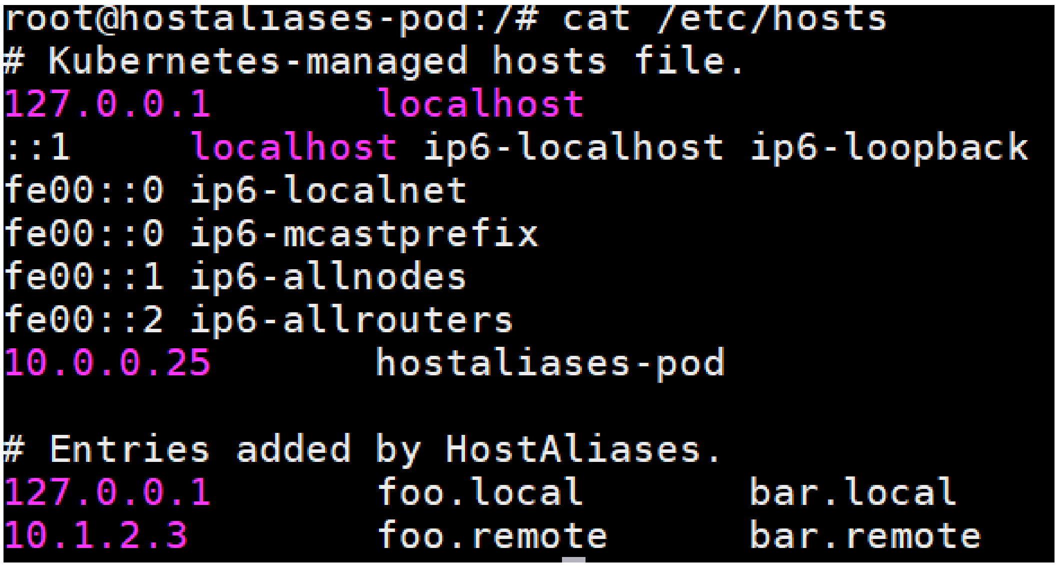
If the pod is in the **Running** state, the pod is successfully created.

```
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

11.5 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.

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.

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 Permission

12.1 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).

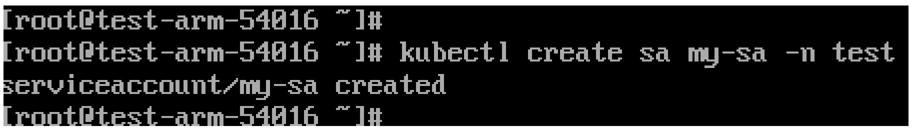
Configuration Method

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.

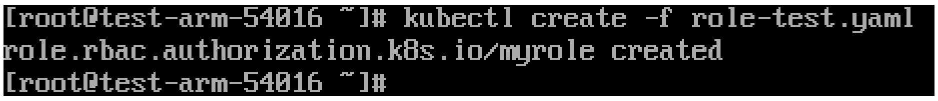
```
vi role-test.yaml
```

The content is as follows:

```
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.

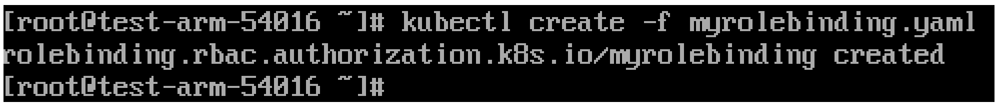
```
vi myrolebinding.yaml
```

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
```



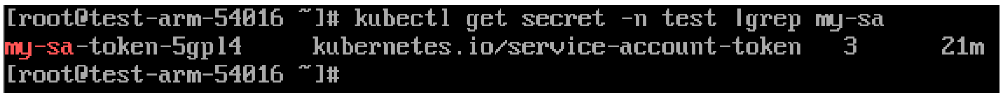
```
[root@test-arm-54016 ~]# kubectl create -f myrolebinding.yaml
rolebinding.rbac.authorization.k8s.io/myrolebinding created
[root@test-arm-54016 ~]#
```

The user information is configured. Now perform **Step 4** to **Step 6** to write the user information to the configuration file.

Step 4 Configure the cluster information.

1. Use the sa name **my-sa** to obtain the secret corresponding to the sa. In the following example, **my-sa-token-z4967** in the first column is the secret name.

```
kubectl get secret -n test |grep my-sa
```

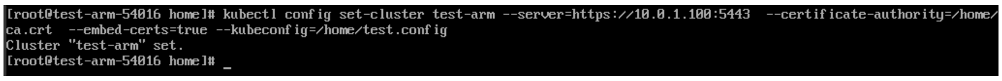


```
[root@test-arm-54016 ~]# kubectl get secret -n test |grep my-sa
my-sa-token-5gpl4      kubernetes.io/service-account-token   3      21m
[root@test-arm-54016 ~]#
```

2. Decrypt the **ca.crt** file in the secret and export it.

```
kubectl get secret my-sa-token-5gpl4 -n test -oyaml |grep ca.crt: | awk '{print $2}' |base64 -d > /home/
ca.crt
```

3. Set the cluster access mode. **test-arm** indicates the cluster to be accessed, **10.0.1.100** indicates the IP address of the API server in the cluster and **/home/test.config** indicates 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://10.0.1.100:5443  --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://10.0.1.100:5443 --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**.

The cluster API server address is an intranet API server address. After an EIP is bound to the cluster, the cluster API server address can also be a public API server address.

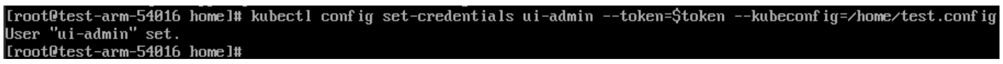
Step 5 Configure the cluster authentication information.

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-5gpl4 -n test | awk '/token:/{print $2}')
```

2. Set the cluster user **ui-admin**.

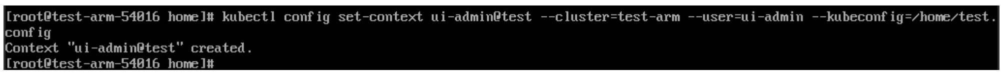
```
kubectl config set-credentials ui-admin --token=$token --kubeconfig=/home/test.config
```



```
[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 6 Configure the context information for cluster authentication. **ui-admin@test** is the context name.

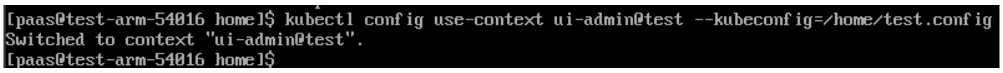
```
kubectl config set-context ui-admin@test --cluster=test-arm --user=ui-admin --kubeconfig=/home/
test.config
```



```
[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 7 Set 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]$
```

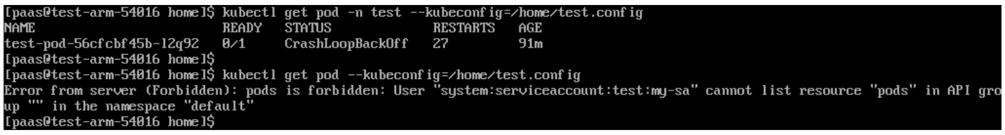
NOTE

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 6**. The user must ensure that the host can access the API server address of the cluster. When performing step **Step 7** on the host and using kubectl, the user must set the kubeconfig parameter to the path of the configuration file.

----End

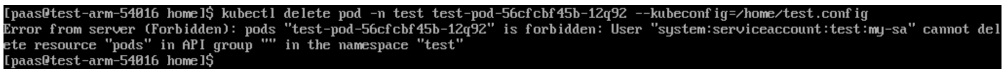
Verification

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-12q92   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 Release

13.1 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.

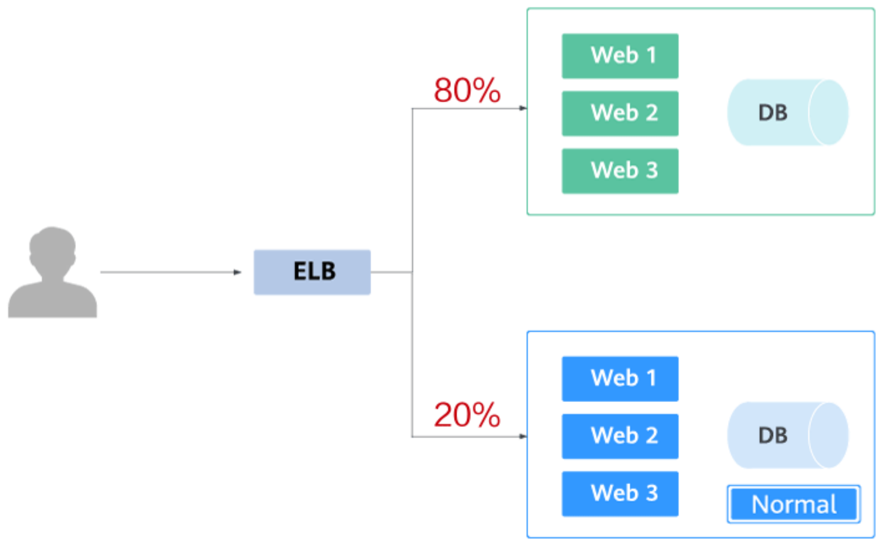
Solution

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.

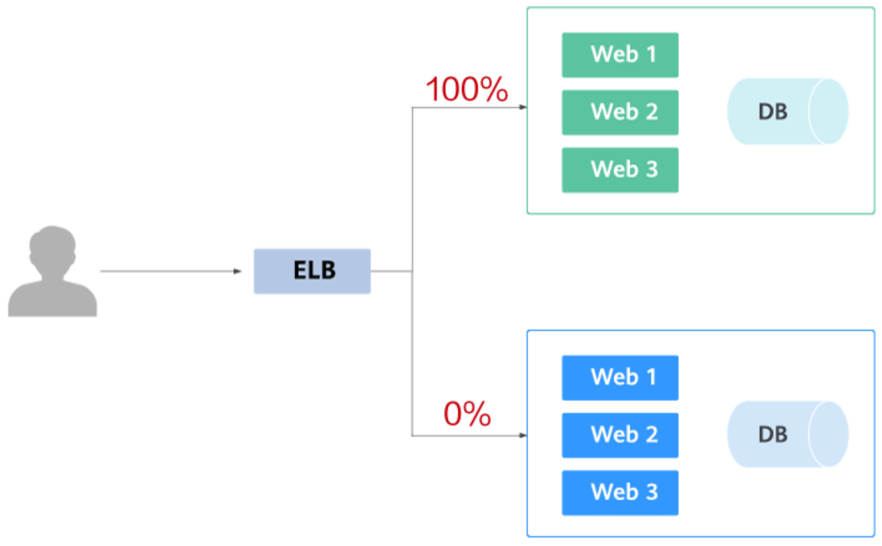
This document describes the principles and practices of grayscale release and blue-green deployment.

- 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.

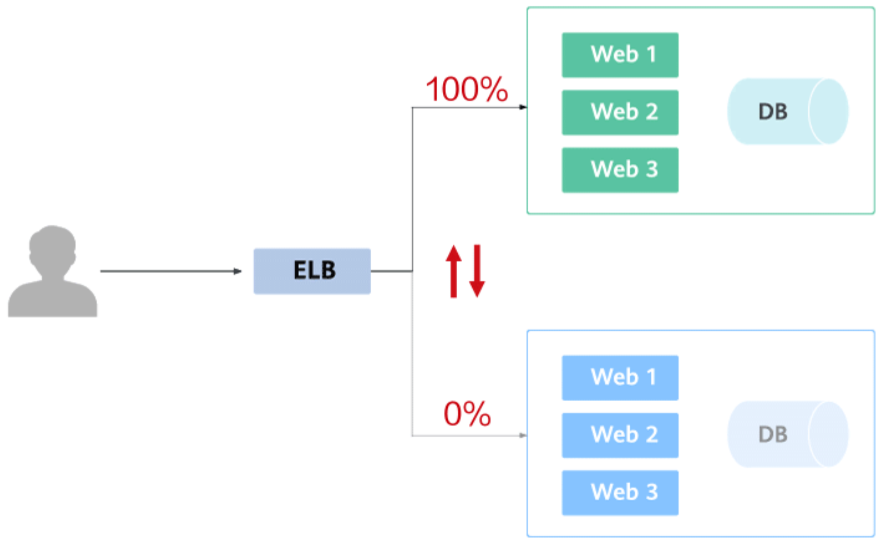
  The following figure shows the general process of grayscale release. First, divide 20% of all service traffic to the new version. If the service version runs normally, gradually increase the traffic proportion and continue to test the performance of the new version. If the new version is stable, switch 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.

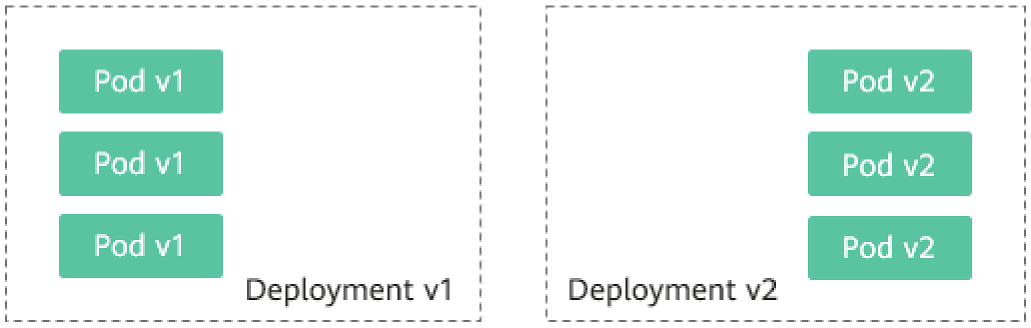


13.2 Using Services to Implement Simple Grayscale Release and Blue-Green Deployment

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

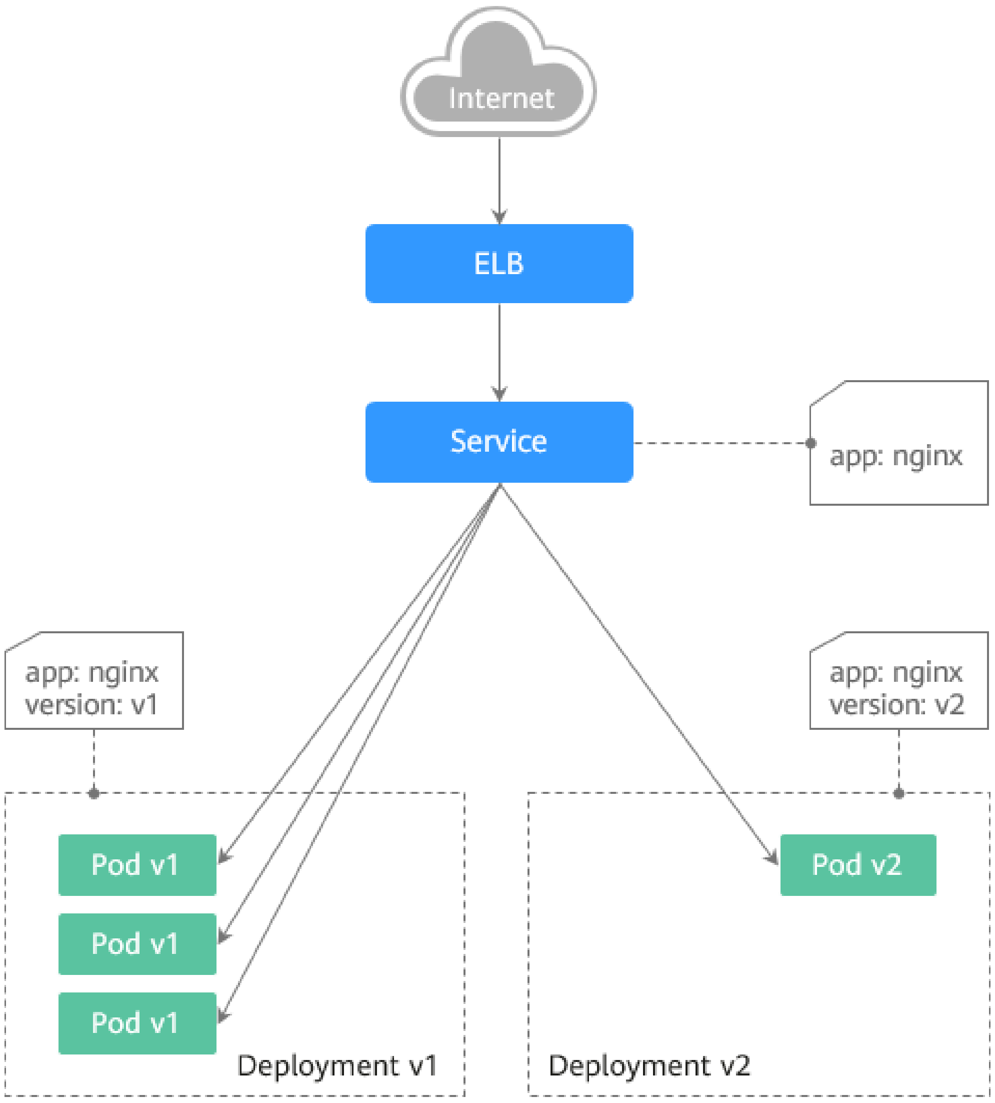
Users usually use Kubernetes objects such as Deployments and StatefulSets to deploy services. Each workload manages a group of pods. The following figure uses Deployment as an example.



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.

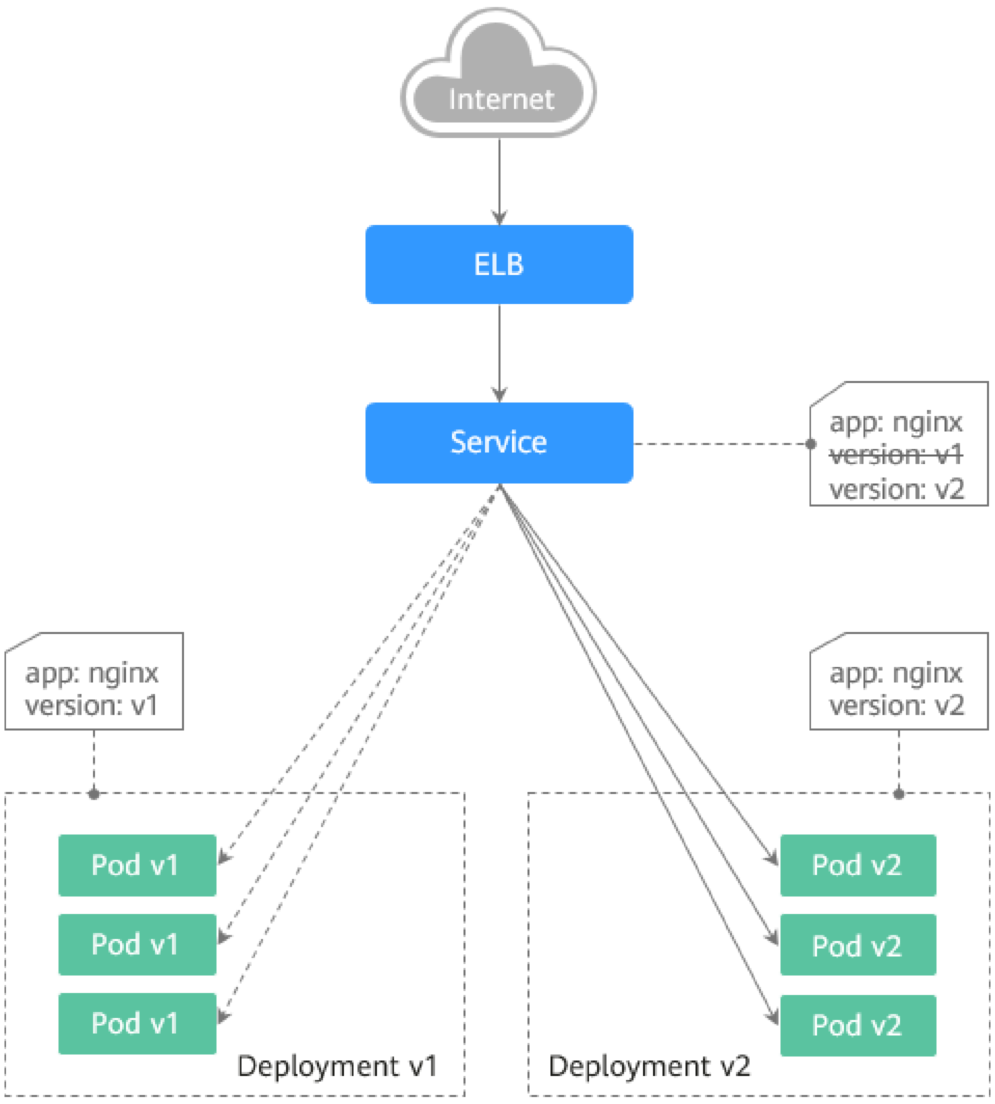
- **Grayscale release principles**

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

13.3 Using Nginx Ingress to Implement Grayscale Release and Blue-Green Deployment

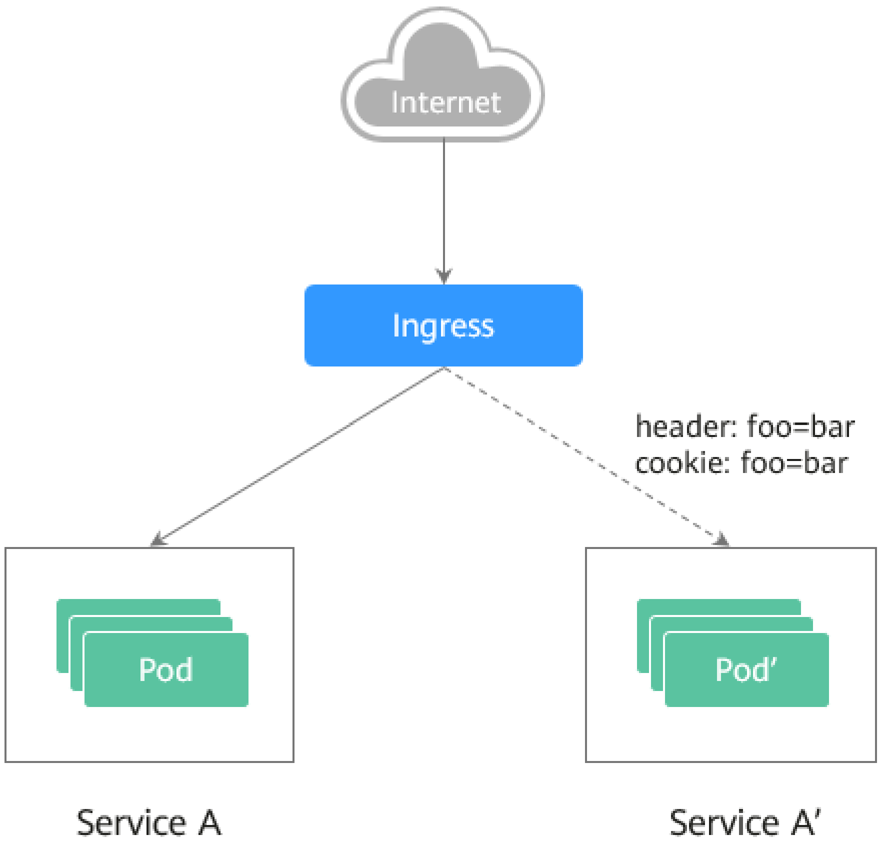
This section describes the scenarios and practices of using Nginx Ingress to implement grayscale release and blue-green deployment.

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:

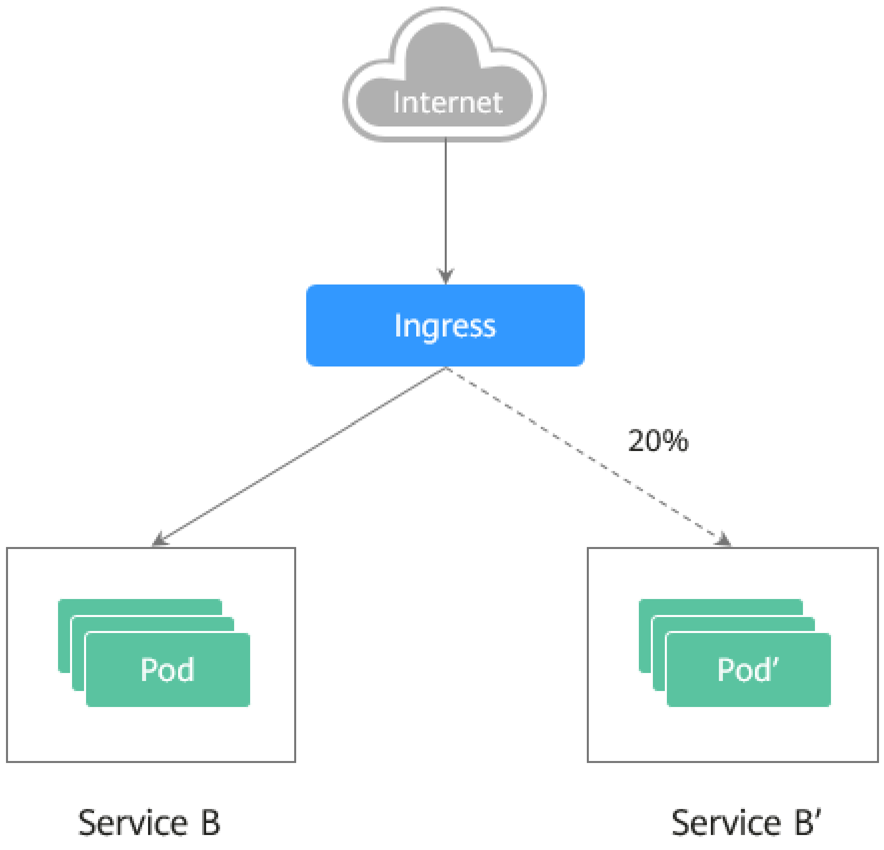
- **Scenario 1: Split some user traffic to the new version.**

  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 .
- 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.