

Cloud Container Engine

Best Practices

Issue 01
Date 2024-11-12



Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2024. All rights reserved.

No part of this document may be reproduced or transmitted in any form or by any means without prior written consent of Huawei Cloud Computing Technologies Co., Ltd.

Trademarks and Permissions



HUAWEI and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

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

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

Contents

1 Checklist for Deploying Containerized Applications in the Cloud.....	1
2 Containerization.....	7
2.1 Containerizing an Enterprise Application (ERP).....	7
2.1.1 Solution Overview.....	7
2.1.2 Procedure.....	10
2.1.2.1 Containerizing an Entire Application.....	10
2.1.2.2 Containerization Process.....	12
2.1.2.3 Analyzing the Application.....	13
2.1.2.4 Preparing the Application Runtime.....	15
2.1.2.5 Compiling a Startup Script.....	17
2.1.2.6 Compiling the Dockerfile.....	17
2.1.2.7 Building and Uploading an Image.....	18
2.1.2.8 Creating a Container Workload.....	19
3 Migration.....	25
3.1 Migrating Kubernetes Clusters to CCE.....	25
3.1.1 Solution Overview.....	25
3.1.2 Planning Resources for the Target Cluster.....	29
3.1.3 Procedure.....	32
3.1.3.1 Migrating Resources Outside a Cluster.....	32
3.1.3.2 Installing the Migration Tool.....	33
3.1.3.3 Migrating Resources in a Cluster.....	37
3.1.3.4 Updating Resources Accordingly.....	40
3.1.3.5 Performing Additional Tasks.....	43
3.1.3.6 Troubleshooting.....	44
4 Disaster Recovery.....	47
4.1 Recommended Configurations for HA CCE Clusters.....	47
4.2 Implementing High Availability for Applications in CCE.....	56
4.3 Implementing High Availability for Add-ons in CCE.....	59
5 Security.....	63
5.1 Configuration Suggestions on CCE Cluster Security.....	63
5.2 Configuration Suggestions on CCE Node Security.....	66
5.3 Configuration Suggestions on CCE Container Runtime Security.....	68

5.4 Configuration Suggestions on CCE Container Security.....	70
5.5 Configuration Suggestions on CCE Container Image Security.....	74
5.6 Configuration Suggestions on CCE Secret Security.....	76
6 Auto Scaling.....	79
6.1 Using HPA and CA for Auto Scaling of Workloads and Nodes.....	79
7 Monitoring.....	87
7.1 Monitoring Multiple Clusters Using Prometheus.....	87
7.2 Reporting Prometheus Monitoring Data to a Third-Party Monitoring Platform.....	91
8 Cluster.....	94
8.1 Suggestions on CCE Cluster Selection.....	94
8.2 Creating a Custom CCE Node Image.....	100
8.3 Connecting to Multiple Clusters Using kubectl.....	106
8.4 Selecting a Data Disk for the Node.....	112
8.5 Protecting a CCE Cluster Against Overload.....	118
9 Networking.....	123
9.1 Planning CIDR Blocks for a Cluster.....	123
9.2 Selecting a Network Model.....	129
9.3 Implementing Sticky Session Through Load Balancing.....	135
9.4 Pre-Binding Container ENI for CCE Turbo Clusters.....	142
9.5 Accessing an IP Address Outside a Cluster That Uses a VPC Network Using Source Pod IP Addresses in the Cluster.....	146
10 Storage.....	149
10.1 Expanding the Storage Space.....	149
10.2 Mounting Object Storage Across Accounts.....	158
10.3 Dynamically Creating an SFS Turbo Subdirectory Using StorageClass.....	171
10.4 Using Custom Storage Classes.....	175
10.5 Scheduling EVS Disks Across AZs Using csi-disk-topology.....	186
11 Container.....	192
11.1 Properly Allocating Container Computing Resources.....	192
11.2 Modifying Kernel Parameters Using a Privileged Container.....	194
11.3 Using Init Containers to Initialize an Application.....	196
11.4 Configuring the /etc/hosts File of a Pod Using hostAliases.....	197
11.5 Locating Container Faults Using the Core Dump File.....	200
12 Permission.....	202
12.1 Configuring kubeconfig for Fine-Grained Management on Cluster Resources.....	202
13 Release.....	206
13.1 Overview.....	206
13.2 Using Services to Implement Simple Grayscale Release and Blue-Green Deployment.....	208
13.3 Using Nginx Ingress to Implement Grayscale Release and Blue-Green Deployment.....	214

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 as needed. <ul style="list-style-type: none"> • Select VPC network or Tunnel network for your CCE standard cluster. • Select Cloud Native Network 2.0 for your CCE Turbo cluster. 	Deployment	After a cluster is created, the network model cannot be changed. Exercise caution when selecting a network model.

Category	Check Item	Type	Impact
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.
	When creating a workload, select a proper access mode (Service). Currently, the following types of Services are supported: ClusterIP, NodePort, and LoadBalancer.	Deployment	Improper Service configuration may cause logic confusion for internal and external access and resource waste.

Category	Check Item	Type	Impact
	When creating a workload, do not set the number of replicas for a single pod. Set a proper node scheduling policy based on your service requirements.	Reliability	For example, if the number of replicas of a single pod is set, the service will be abnormal when the node or pod is abnormal. To ensure that your pods can be successfully scheduled, ensure that the node has idle resources for container scheduling after you set the scheduling rule.
	Properly set affinity and anti-affinity.	Reliability	If affinity and anti-affinity are both configured for an application that provides Services externally, Services may fail to be accessed after the application is upgraded or restarted.
	When creating a workload, set the pre-stop processing command (Lifecycle > Pre-Stop) to ensure that the services running in the pods can be completed in advance in the case of application upgrade or pod deletion.	Reliability	If the pre-stop processing command is not configured, the pod will be directly killed and services will be interrupted during application upgrade.

Table 1-2 Data reliability

Category	Check Item	Type	Impact
Container data persistency	Select a proper data volume type based on service requirements.	Reliability	When a node is faulty and cannot be recovered, data in the local disk cannot be recovered. Therefore, you are advised to use cloud storage volumes to ensure data reliability.
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.

Category	Check Item	Type	Impact
	<p>Do not modify information about resources created by CCE, such as security groups and EVS disks. Resources created by CCE are labeled cce.</p>	Deployment	CCE cluster functions may be abnormal.
Proactive O&M	<p>CCE provides multi-dimensional monitoring and alarm reporting functions, allowing users to locate and rectify faults as soon as possible.</p> <ul style="list-style-type: none"> • 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	<p>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.</p>

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.	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • 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
<p>Method 2: Containerize first the application components that are frequently updated or have high requirements on auto scaling.</p>	<ul style="list-style-type: none"> ● 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. 	<p>Need to decouple some services.</p>

Containerization Mode	Advantage	Disadvantage
<p>Method 3: Transform an application to microservices and then containerize the microservices one by one.</p>	<ul style="list-style-type: none"> ● 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. 	<p>Need to transform the application to microservices, which involves a large number of changes.</p>

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.

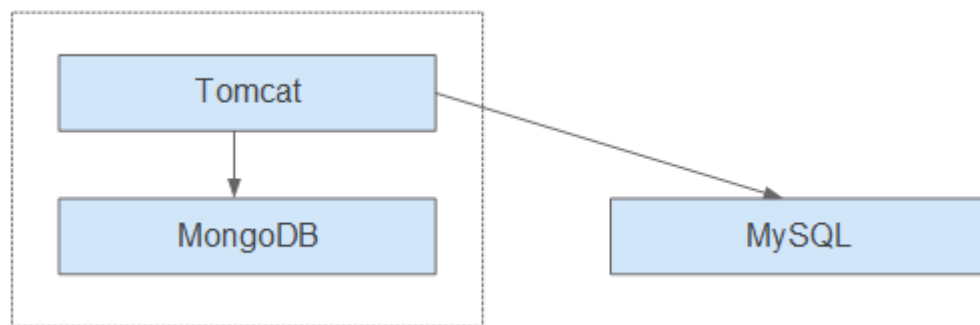
No recoding or re-architecting is required. You only need to pack the entire application into a container image and deploy the container image on CCE.

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.

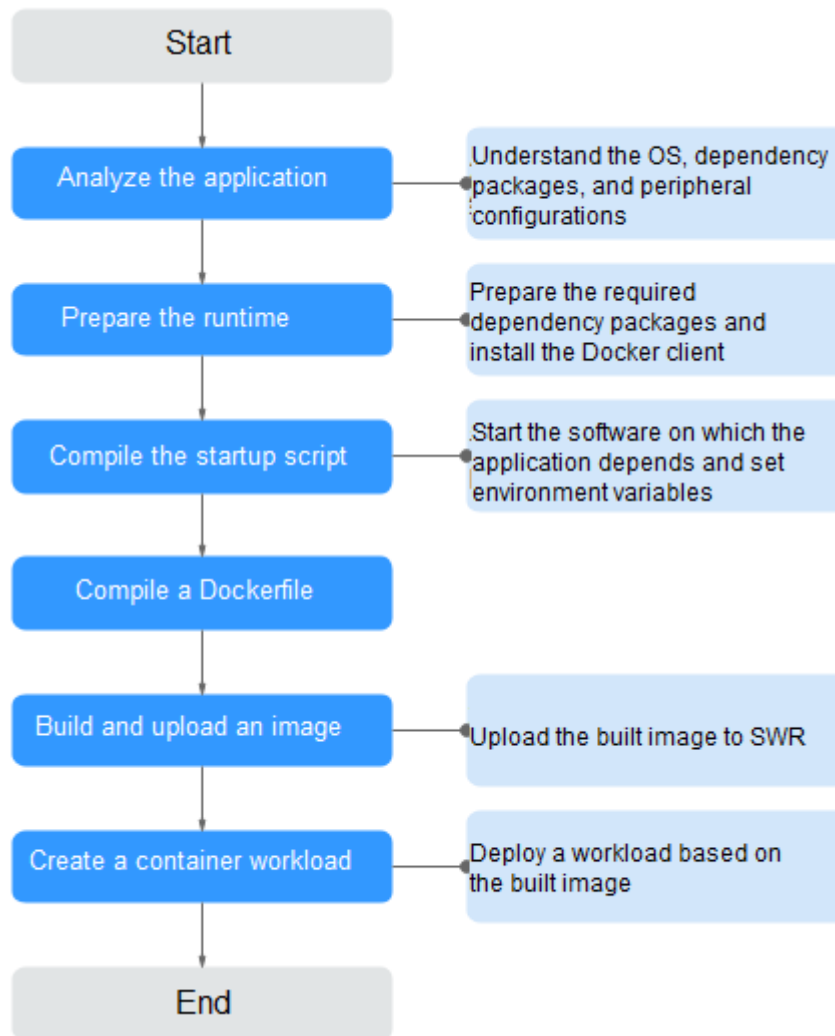
2.1.2.2 Containerization Process

To fully containerize an application, you must go through the entire process.

This involves analyzing the application, setting up the runtime environment for the application, compiling the startup script and Dockerfile, creating and uploading images, and creating containerized workloads.

For details about each step of the containerization, see [Containerization Process](#).

Figure 2-2 Process of containerizing an application



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	<p>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.</p> <p>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.</p>
	Dependency package	<p>Understand required dependency packages, such as OpenSSL and other system software, and their version numbers.</p> <p>In this example, no dependency package is required.</p>
Deployment mode	Peripheral configurations	<p>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.</p>
		<p>External services with which the application needs to interconnect, such as databases and file systems.</p> <p>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.</p> <p>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.</p> <pre>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</pre>

Category	Sub-category	Description
	Application configurations	<p>Sort out the configuration parameters, such as configurations that need to be modified frequently and those remain unchanged during the running of the application.</p> <p>In this example, no application configurations need to be extracted.</p> <p>NOTE To avoid frequent image replacement, you are advised to classify configurations of the application.</p> <ul style="list-style-type: none"> 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 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.

It is recommended that you install Docker and build images as the user **root**. Make sure to obtain the user **root** password for the host where Docker will be installed beforehand.

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 check the Docker version:

```
docker version
```

```
Client:
Version: 17.12.0-ce
API Version:1.35
```

```
...
```

Version indicates the version number.

----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 command to switch to the directory where the application is to be stored:

```
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
```

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:v1 .
```

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	<p>Create a VPC before you create a cluster. A VPC provides an isolated, configurable, and manageable virtual network environment for CCE clusters.</p> <p>If you have a VPC already, skip to the next task.</p> <ol style="list-style-type: none"> 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.
2	Creating a key pair	<p>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.</p> <ol style="list-style-type: none"> 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 host the private key of the key pair, and I have read and agree to the Key Pair Service Disclaimer, and click OK. 5. View and save the private key. For security purposes, a key pair can be downloaded only once. Keep it secure to ensure successful login.

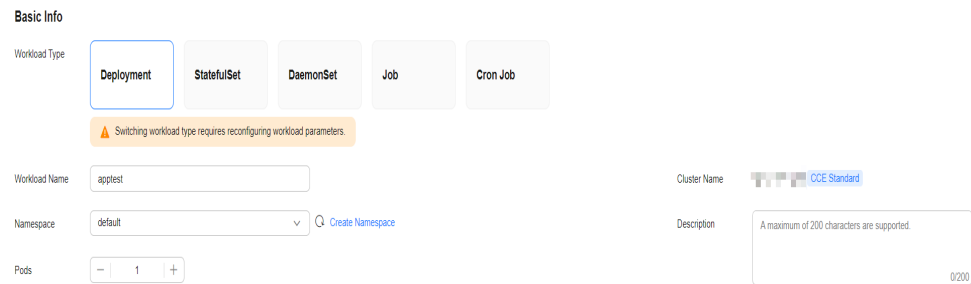
Step 2 Create a cluster and a node.

1. Log in to the CCE console. On the **Clusters** page, click **Buy Cluster** and select the type for the cluster to be created.
Configure cluster parameters and select the VPC created in [Step 1](#).
2. Buy a node and select the key pair created in [Step 1](#) as the login option.

Step 3 Deploy a workload on CCE.

1. Log in to the CCE console and click the name of the cluster to access the cluster console. In the navigation pane, choose **Workloads** and click **Create Workload**.
2. Configure the following parameters, and retain the default settings for other parameters:
 - **Workload Name:** Set it to **apptest**.
 - **Pods:** Set it to **1**.

Figure 2-3 Basic settings



- In the **Container Settings** area, select the image uploaded in **Building and Uploading an Image**.
- 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.

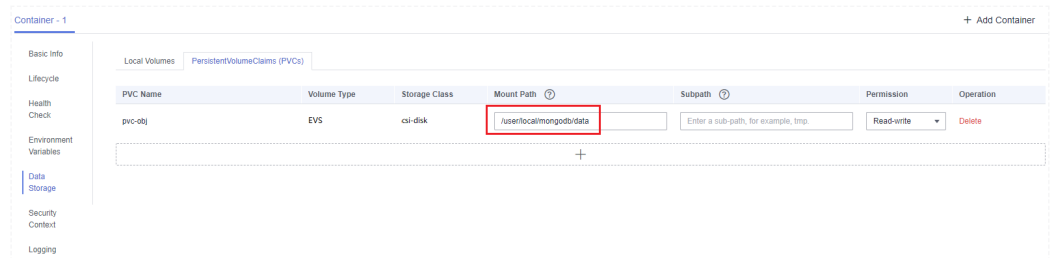
- In the **Container Settings** area, choose **Data Storage** and configure cloud storage for persistent data storage.

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



- 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:** 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.

Figure 2-5 Creating a Service

Create Service ×

Service Name:

Service Type:

- ClusterIP**
Expose services through the internal IP of the cluster, which can only be accessed within the cluster
- NodePort** (Selected)
Expose services via IP and static port (NodePort) on each node
- LoadBalancer**
Provide external services through ELB load balancing, high availability, ultra-high performance, stability and security
- DNAT**
Expose cluster node access type services through NAT gateway, support multiple nodes to share and use elastic IP

i It is recommended to select the load balancing access type for out-of-cluster access

Service Affinity: Cluster-level Node-level ⓘ

Ports:

Protocol	Container Port...	Service Port ⓘ	Node Port	O...
TCP	8080	80	Auto	Delete
+				

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 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

- CCE clusters that are no longer maintained and cannot be upgraded in place

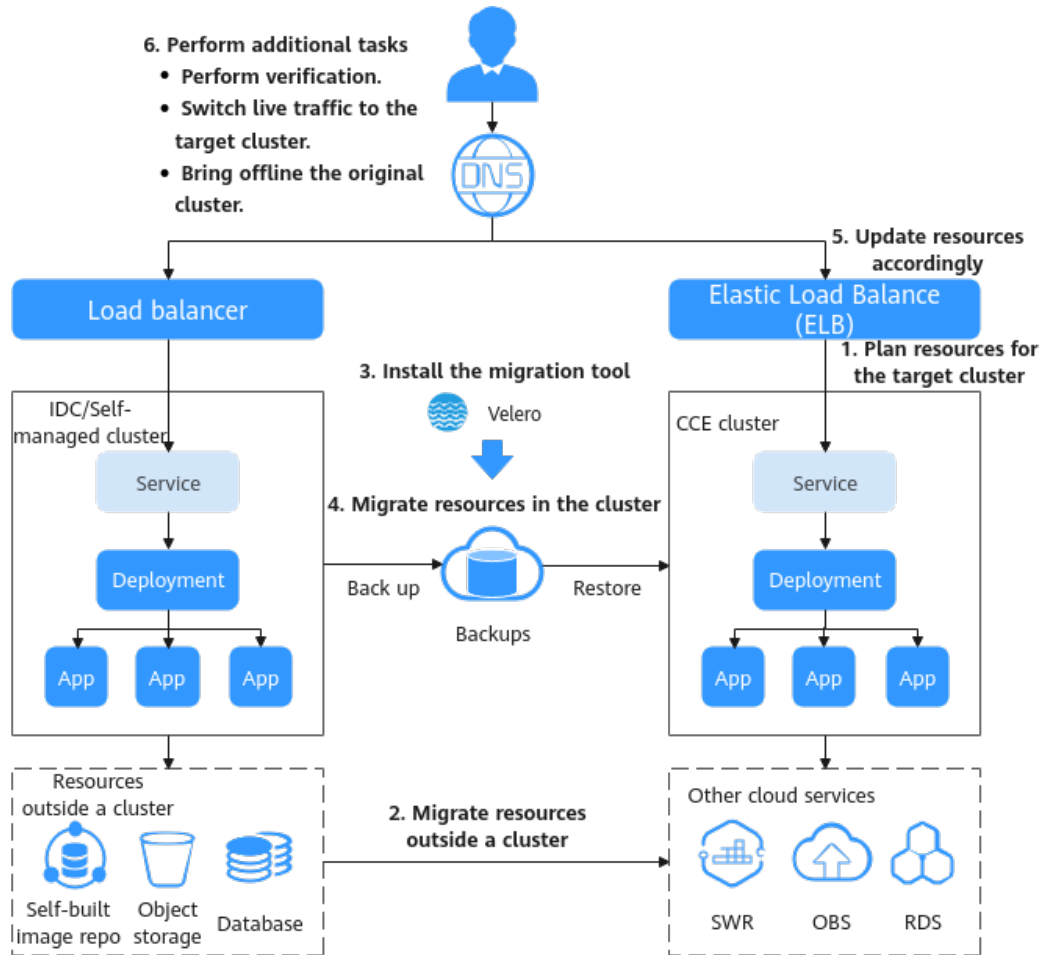
Before the migration, analyze all resources in the source clusters and then determine the migration solution. Resources that can be migrated include resources inside and outside the clusters, as listed in the following table.

Table 3-1 Resources that can be migrated

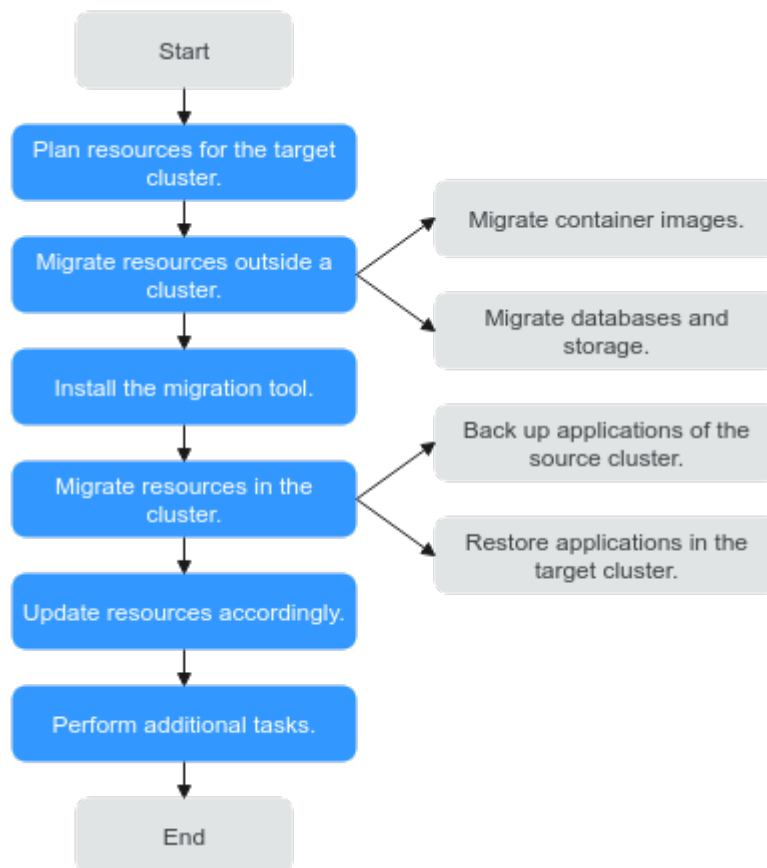
Category	Migration Object	Remarks
Resources inside a cluster	All objects in a cluster, including pods, jobs, Services, Deployments, and ConfigMaps.	<p>You are not advised to migrate the resources in the velero and kube-system namespaces.</p> <ul style="list-style-type: none"> • velero: Resources in this 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. <p>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.</p>
	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



Migration Process



The cluster migration process is as follows:

Step 1 Plan resources for the target cluster.

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

Step 2 Migrate resources outside a cluster.

To migrate resources outside the cluster, see [Migrating Resources Outside a Cluster](#).

Step 3 Install the migration tool.

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

Step 4 Migrate resources in the cluster.

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

- **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	<ul style="list-style-type: none"> ● 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	<ul style="list-style-type: none"> ● 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	<ul style="list-style-type: none"> ● 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	<ul style="list-style-type: none"> ● EulerOS ● CentOS 	EulerOS

Resource	Key Performance Parameter	Description	Example Value
	Node Specifications (vary depending on the actual region)	<ul style="list-style-type: none"> ● 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	<ul style="list-style-type: none"> ● 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	<ul style="list-style-type: none"> ● 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 Procedure

3.1.3.1 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

3.1.3.2 Installing the Migration Tool

Velero is an open-source backup and migration tool for Kubernetes clusters. With Restic's PV data backup capability integrated into it, Velero can back up Kubernetes resource objects (such as Deployments, jobs, Services, and ConfigMaps) in source clusters and data in PVs mounted to pods and uploaded them to object storage. When a disaster occurs or migration is required, a target cluster can obtain the corresponding backup data from the object storage using Velero and restore cluster resources as required.

According to [Migration Solution](#), prepare temporary object storage to store backup files before the migration. Velero supports OBS or [MinIO](#) as the object storage. The object storage requires sufficient storage space for storing backup files. You can estimate the storage space based on your cluster scale and data volume. OBS buckets are recommended for data 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.

(Optional) Installing MinIO

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

MinIO can be installed in any of the following locations:

- Temporary ECS outside the cluster
If the MinIO server is installed outside the cluster, backup files will not be affected when a catastrophic fault occurs in the cluster.
- Idle nodes in the cluster
You can remotely log in to a node and install MinIO or install the containerized MinIO. For details, see [Velero official document](#).

NOTICE

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

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

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

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

Step 1 Download MinIO.

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

Step 2 Set the username and password of MinIO.

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

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

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

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

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

 **NOTE**

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

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

----End

Installing Velero

Go to the OBS console or MinIO console and create a bucket named **velero** to store backup files. You can custom the bucket name, which must be used when installing Velero. Otherwise, the bucket cannot be accessed and the backup fails. For details, see [Step 5](#).

NOTICE

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

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

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

Step 2 Download the binary file of Velero 1.13.1.

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

Step 3 Install the Velero client.

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

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

```
vim credentials-velero
```

Replace the AK/SK in the file based on the site requirements. If MinIO is used, the AK/SK are the username and password created in [Step 2](#).

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

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

Table 3-3 Installation parameters of Velero

Parameter	Description
--provider	AWS S3 component to be used
--plugins	API component compatible with AWS S3. Both OBS and MinIO support the S3 protocol.
--bucket	Name of the object storage bucket for storing backup files. The bucket must be created in advance.
--secret-file	Secret file for accessing the object storage, that is, the credentials-velero file created in Step 4 .
--use-node-agent	Whether to enable PV data backup. You are advised to enable this function. Otherwise, storage volume resources cannot be backed up.
--use-volume-snapshots	Whether to create the VolumeSnapshotLocation object for PV snapshot, which requires support from the snapshot program. Set this parameter to false .
--backup-location-config	OBS bucket configurations, including region, s3ForcePathStyle, and s3Url.
region	Region to which object storage bucket belongs. <ul style="list-style-type: none"> 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. <ul style="list-style-type: none"> 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. <p>NOTE</p> <ul style="list-style-type: none"> The access port in s3Url must be set to the API port of MinIO instead of the console port. The default API port of MinIO is 9000. To access MinIO installed outside the cluster, enter the public IP address of MinIO.

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


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

 **NOTE**

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

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

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

----End

3.1.3.3 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 images can be properly pulled after cluster migration, migrate the images to SWR.
- CCE does not support EVS disks of the **ReadWriteMany** type. If resources of this type exist in the source cluster, change the storage type to **ReadWriteOnce**.
- Velero cannot back up or restore HostPath volumes. For details, see [Limitations](#). To back up storage volumes of this type, replace the hostPath volumes with local volumes. 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:

```
kubectrl -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.

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

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

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

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

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

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

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

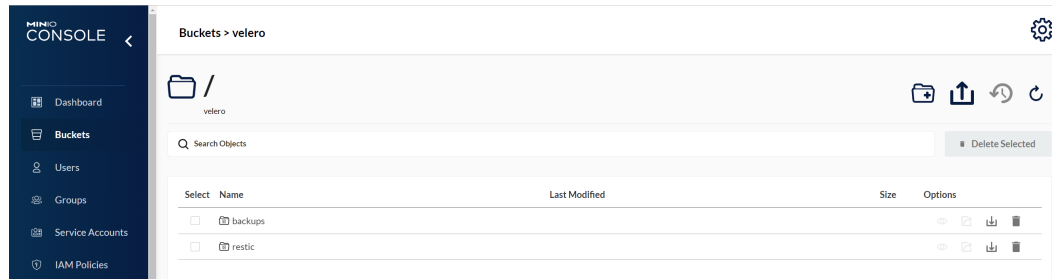
Step 3 Check the backup status.

```
velero backup get
```

Information similar to the following is displayed:

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

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



----End

Restoring Applications in the Target Cluster

The storage infrastructure of an on-premises cluster is different from that of a cloud cluster. After the cluster is migrated, PVs cannot be mounted to pods. Therefore, during the migration, update the storage class of the target cluster to shield the differences of underlying storage interfaces between the two clusters when creating a workload and request storage resources of the corresponding type. For details, see [Updating the Storage Class](#).

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

3.1.3.4 Updating Resources Accordingly

Updating Images

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

Step 1 Migrate the image resources to SWR.

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:

```
kubectrl 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 kubectrl.

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

```
kubectrl 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 subnet where the load balancer resides
```

```
kubernetes.io/elb.session-affinity-mode: SOURCE_IP # Enable the sticky session based on the source IP address.
```

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

----End

Updating the Storage Class

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

NOTICE

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

Method 1: Creating a ConfigMap mapping

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

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

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

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

----End

Method 2: Creating a storage class with the same name

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

```
kubectl get sc
```

Information similar to the following is displayed:

NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBINDINGMODE	ALLOWVOLUMEEXPANSION	AGE
csi-disk	everest-csi-provisioner	Delete	Immediate	true	3d23h
csi-disk-topology	everest-csi-provisioner	Delete	WaitForFirstConsumer	true	3d23h
csi-sfs	everest-csi-provisioner	Delete	Immediate	false	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-sfs	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.

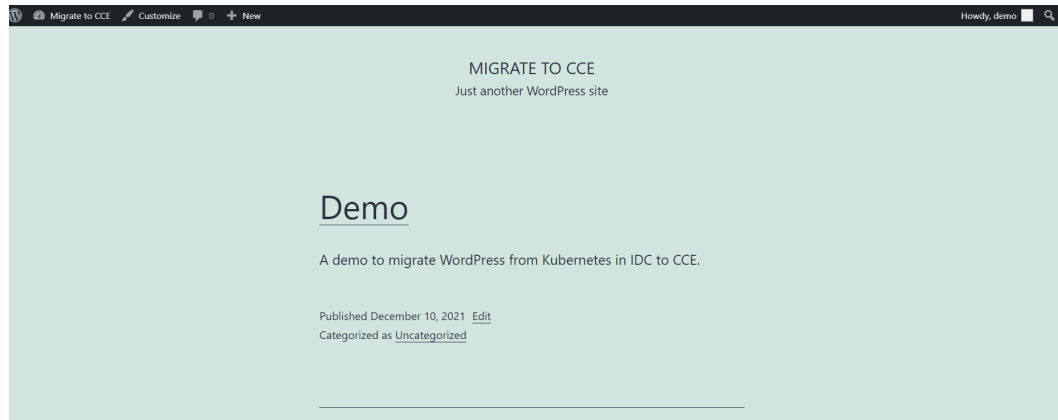
----End

3.1.3.5 Performing Additional Tasks

Verifying Application Functions

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

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



Switching Live Traffic to the Target Cluster

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

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

Bringing the Source Cluster Offline

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

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

3.1.3.6 Troubleshooting

Storage Volumes of the HostPath Type Cannot Be Backed Up

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

NOTE

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

Step 1 Create a storage class for the Local volume.

Example YAML:

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


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

Example YAML:

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

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

```
kubectl get pv
```

Information similar to the following is displayed:

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

----End

Backup Tool Resources Are Insufficient

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

Before installing Velero:

You can specify the size of resources used by Velero and Restic when [installing Velero](#).

The following is an example of installation parameters:

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

After Velero is installed:

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

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

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

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

----End

4 Disaster Recovery

4.1 Recommended Configurations for HA CCE Clusters

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

Item	Description	Recommended Operations
Master node	CCE is a hosted Kubernetes cluster service. You do not need to perform O&M on the master nodes. You can configure your cluster specifications to improve the stability and reliability.	<ul style="list-style-type: none"> • Deploying the Master Nodes in Different AZs • Selecting a Network Model • Selecting a Service Forwarding Mode • Configuring Quotas and Limits for the Cloud Service Resources and Resources in a Cluster • Monitoring Metrics of the Master Nodes
Worker node	In a Kubernetes cluster, the data plane consists of worker nodes that can run containerized applications and transmit network traffic. When using CCE, perform O&M on worker nodes by yourself. To achieve HA, ensure the worker nodes' scalability and repairability and pay attention to the running statuses of the worker nodes' key components.	<ul style="list-style-type: none"> • Partitioning Data Disks Attached to a Node • Running npd • Configuring the DNS Cache • Properly Deploying CoreDNS

Item	Description	Recommended Operations
Application	If you want your applications to be always available, especially during peak hours, run them in a scalable and elastic manner and pay attention to their running statuses.	<ul style="list-style-type: none"> ● Running Multiple Pods ● Configuring Resource Quotas for a Workload ● Deploying an Application in Multiple AZs ● Deploying the System Add-ons in Multiple AZs ● Configuring Auto Scaling ● Viewing Logs, Monitoring Metrics, and Adding Alarm Rules

Deploying the Master Nodes in Different AZs

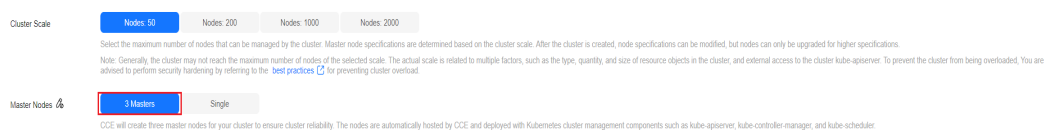
Multiple regions are provided for you to deploy your services, and there are different availability zones (AZs) in each region. An AZ is a collection of one or more physical data centers with independent cooling, fire extinguishing, moisture-proof, and electricity facilities in each AZ. AZs within a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

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

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

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

Figure 4-1 Configuring an HA cluster



Selecting a Network Model

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

For details, see [Planning CIDR Blocks for a Cluster](#).

Selecting a Service Forwarding Mode

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

CCE supports the iptables and IPVS forwarding modes.

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

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

CCE allows you to configure resource quotas and limits for your cloud service resources and resources in your clusters. This prevents excessive use of resources. When creating your applications for CCE clusters, consider these limits and

periodically review them. This will avoid scaling failures caused by insufficient quotas during application running.

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

Monitoring Metrics of the Master Nodes

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

To monitor the kube-apiserver, kube-controller, kube-scheduler, and etcd-server components of the master nodes, you need to install the [Cloud Native Cluster Monitoring](#) add-on in the cluster. With grafana, you can use the [Kubernetes monitoring overview dashboard](#) to monitor metrics of Kubernetes API server requests and latency and etcd latency.

If your prometheus add-on is used, you can manually add monitoring metrics. For details, see [Monitoring Metrics of Master Node Components Using Prometheus](#).

Partitioning Data Disks Attached to a Node

By default, the first data disk of a worker node is for storing the container runtime and kubelet components. The remaining capacity of this data disk affects image download and container startup and running. For details, see [Data Disk Space Allocation](#).

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

Running npd

A failure in a worker node may affect the availability of the applications. [CCE Node Problem Detector](#) is used to monitor node exceptions. It helps you detect and handle latent exceptions in a timely manner. You can also customize the check items, including target node, check period, and triggering threshold.

Configuring the DNS Cache

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

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

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

You can install the [NodeLocal DNSCache](#) add-on to deploy NodeLocal DNSCache. For details, see [Using NodeLocal DNSCache to Improve DNS Performance](#).

Properly Deploying CoreDNS

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

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

Running Multiple Pods

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

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

Using Containers to Isolate Processes

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

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

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

Configuring Resource Quotas for a Workload

Configure and adjust resource requests and limits for all workloads.

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

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

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

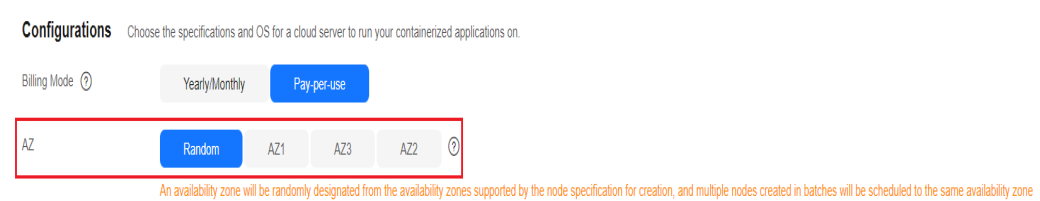
Additionally, the resource usage must be claimed for all pods. For a pod that does not claim the resource usage, after it is scheduled to a node, Kubernetes does not deduct the resources used by this pod from the node on which it is running. Other pods may still be scheduled to this node.

Deploying an Application in Multiple AZs

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

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

Figure 4-2 Specifying an AZ of a node



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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
```



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

You can also use [Pod Topology Spread Constraints](#) to deploy pods in multiple AZs.

Deploying the System Add-ons in Multiple AZs

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

Table 4-1 Deployment description

Mode	Configuration Description	Usage Description	Recommended Configuration Scenario
Preferred	Add-on pods will have labels with the key topology.kubernetes.io/zone for soft anti-affinity deployment, and the anti-affinity type is preferredDuringSchedulingIgnoredDuringExecution .	Add-on pods will be preferentially scheduled to nodes in different AZs. If resources in some AZs are insufficient, some add-on pods may be scheduled to the same AZ which has enough resources.	No mandatory requirements for multi-AZ DR

Mode	Configuration Description	Usage Description	Recommended Configuration Scenario
Required	Add-on pods will have labels with the key topology.kubernetes.io/zone for hard anti-affinity deployment, and the anti-affinity type is requiredDuringSchedulingIgnoredDuringExecution .	A maximum of one pod of the same add-on can be deployed in each AZ. The number of running pods cannot exceed the number of AZs in the cluster. If the node where the add-on pod runs is faulty, pods running on the faulty node cannot be automatically migrated to other nodes in the same AZ.	Changing number of AZs (This mode is used to prevent all pods from being scheduled to the node in the current AZ in advance.)
Equivalent mode	Add-on pods will have labels with the key topology.kubernetes.io/zone for configuring topology spread constraints. The pod difference between different topology domains cannot exceed 1 for add-on pods to be evenly distributed in different AZs.	The effect of this mode is between that of the preferred mode and that of the required mode. In the equivalent mode, add-on pods can be deployed in different AZs. Additionally, multiple pods can be deployed in a single AZ when there are more pods than AZs. To use this mode, you need to plan node resources in each AZ in advance to ensure that each AZ has enough node resources for deploying pods. (If there are more than one add-on pods in a single AZ, the nodes to which the add-on pods can be scheduled in each AZ should be one more than the actual add-on pods in the current AZ.) This ensures successful deployment of add-on pods although node resources in some AZ are insufficient and smooth scheduling of add-on pods during update.	Scenarios have high requirements for DR

Configuring Health Check for a Container

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

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

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

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

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

For details, see [Configuring Container Health Check](#).

Configuring Auto Scaling

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

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

- **Workload scaling:** When deploying applications in pods, you can configure requested resources and resource limits for the pods to prevent unlimited usage of resources during peak hours. However, after the upper limit is reached, an application error may occur. To resolve this issue, scale in the number of pods to share workloads.
- **Node scaling:** After the number of pods grows, the resource usage of the node may increase to a certain extent. This results in that the added pods cannot be scheduled. To solve this problem, scale in or out nodes based on the resource usage.

For details, see [Using HPA and CA for Auto Scaling of Workloads and Nodes](#).

Viewing Logs, Monitoring Metrics, and Adding Alarm Rules

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

4.2 Implementing High Availability for Applications in CCE

Basic Principles

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

1. Deploy three master nodes for the cluster.

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

Procedure

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

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

Create workloads according to the following podAntiAffinity rules:

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

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 250m
```

```

    memory: 512Mi
    requests:
      cpu: 250m
      memory: 512Mi
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 50
          podAffinityTerm:
            labelSelector:           # Select the label of the workload to realize the anti-affinity
            between this container and the workload.
            matchExpressions:
              - key: app
                operator: In
                values:
                  - nginx
            namespaces:
              - default
            topologyKey: topology.kubernetes.io/zone # It takes effect in the same AZ.
        - weight: 50
          podAffinityTerm:
            labelSelector:           # Select the label of the workload to realize the anti-affinity
            between this container and the workload.
            matchExpressions:
              - key: app
                operator: In
                values:
                  - nginx
            namespaces:
              - default
            topologyKey: kubernetes.io/hostname # It takes effect on the node.
  imagePullSecrets:
    - name: default-secret

```

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

```

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

```

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

```

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

```

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

```

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

```

4.3 Implementing High Availability for Add-ons in CCE

Application Scenarios

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

As add-ons have become more popular, their stability and reliability have become essential requirements. By default, CCE implements a policy for add-on deployment where worker nodes have a hard anti-affinity configuration, and AZs have a soft anti-affinity configuration. This section explains how to enhance the CCE add-on scheduling policy, allowing you to customize the deployment policy according to your requirements.

Deployment Solution

An add-on typically runs as Deployments or DaemonSets. By default, DaemonSet pods are deployed on all nodes. To ensure HA of the add-on, there are multiple pods, AZ affinity policies, and specified node scheduling configured for Deployments.

Pod-level HA solution:

- **Increasing the Number of Pods:** Multi-pod deployment can effectively prevent service unavailability caused by a single point of failure (SPOF).

Node-level HA solutions:

- **Deploying the Add-on Pods on Dedicated Nodes:** To prevent resource preemption between service applications and core add-ons, it is best to deploy the core add-on pods on dedicated nodes. This ensures that the add-on resources are isolated and restricted on the node level.
- **Deploying the Add-on in Multiple AZs:** Multi-AZ deployment can effectively prevent service unavailability caused by the failure of a single AZ.

Take the CoreDNS add-on as an example. This add-on is deployed as two pods by default in the preferred mode, and the scheduling policies are hard anti-affinity for nodes and soft anti-affinity for AZs. In this case, two nodes are needed to ensure that all add-on pods in the cluster can run properly, and Deployment pods of the add-on can be preferentially scheduled to nodes in different AZs.

The following sections describe how to further improve the add-on SLA.

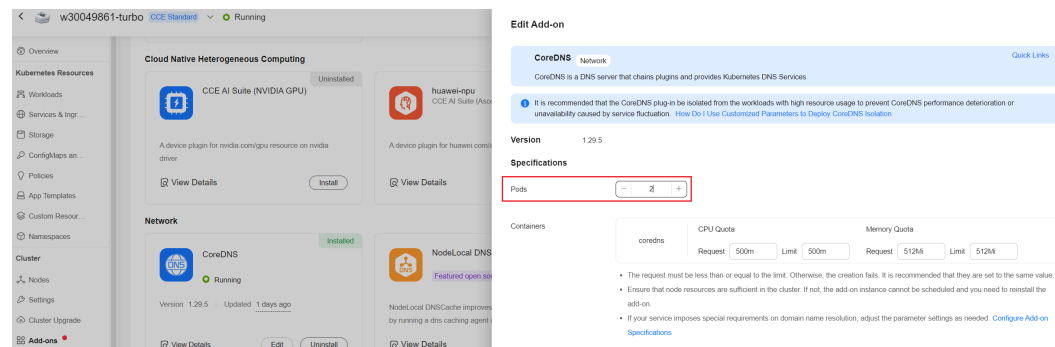
Increasing the Number of Pods

You can adjust the number of CoreDNS pods ensure high performance and HA.

- Step 1** Log in to the CCE console and click the cluster name to access the cluster console. In the navigation pane, choose **Add-ons**, locate **CoreDNS** on the right, and click **Edit**.

Step 2 Increase the number of replicas.

Figure 4-3 Changing the pod quantity



Step 3 Click **OK**.

----End

Deploying the Add-on Pods on Dedicated Nodes

You can adjust the node affinity policy of CoreDNS and make the CoreDNS pods run on dedicated nodes. This can prevent the CoreDNS resources from being preempted by service applications.

A custom policy is used as an example.

Step 1 Log in to the CCE console and click the cluster name to access the cluster console. In the navigation pane, choose **Nodes**.

Step 2 Click the **Nodes** tab, select the node dedicated for CoreDNS, and click **Labels and Taints** above the node list.

Add the following labels:

- Key: **node-role.kubernetes.io/coredns**
- Value: **true**

Add the following taints:

- Key: **node-role.kubernetes.io/coredns**
- Value: **true**
- Effect: **NoSchedule**

Step 3 In the navigation pane, choose **Add-ons**, locate **CoreDNS**, and click **Edit**.

Step 4 Select Custom Policies for **Node Affinity** and add the preceding node label.

Add tolerations for the preceding taint.

Step 5 Click **OK**.

----End

Deploying the Add-on in Multiple AZs

By default, the add-on scheduling policy can handle single-node faults. However, if your services require a higher SLA, you can create nodes with different AZ

specifications on the node pool page and set the multi-AZ deployment mode of the add-on to the required mode.

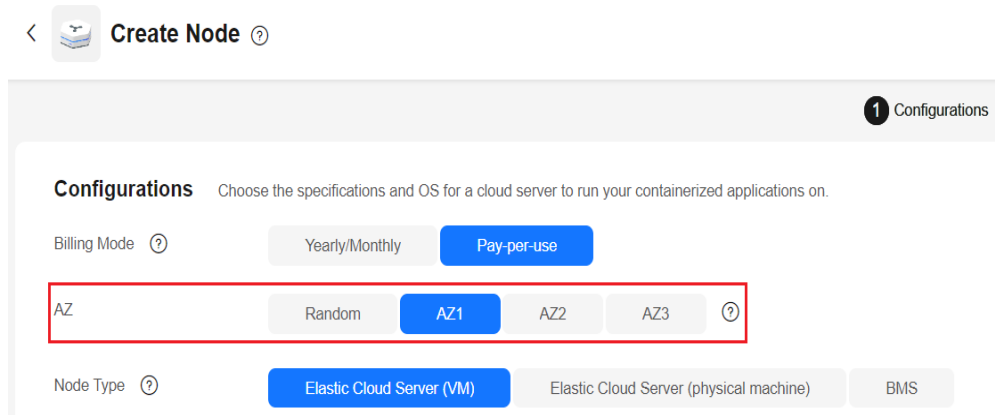
Step 1 Log in to the CCE console and click the cluster name to access the cluster console.

Step 2 Create nodes in different AZs.

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

1. In the navigation pane, choose **Nodes**, click the **Nodes** tab, and click **Create Node** in the upper right corner.
2. On the page displayed, select an AZ for the node.

Figure 4-4 Creating a node

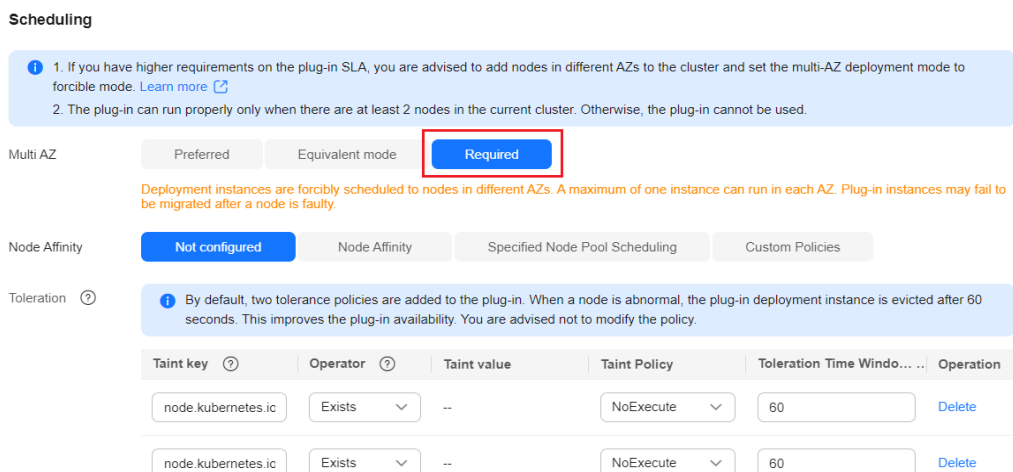


3. Configure other mandatory parameters following instructions to complete the creation.

Step 3 In the navigation pane, choose **Add-ons**. In the right pane, locate **CoreDNS** and click **Edit**.

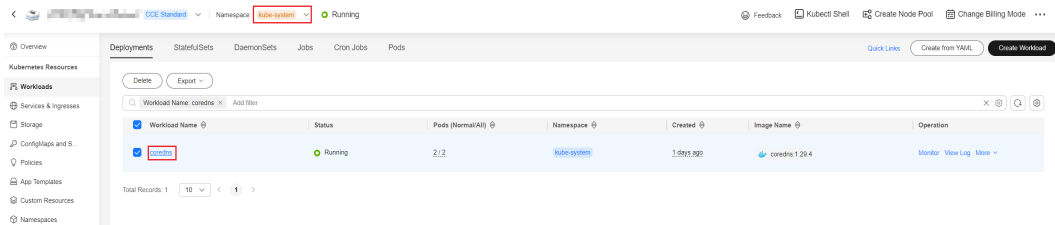
Step 4 In the window that slides out from the right, set **Multi AZ** to **Required** and click **Install**.

Figure 4-5 Changing the multi-AZ deployment mode to the required mode



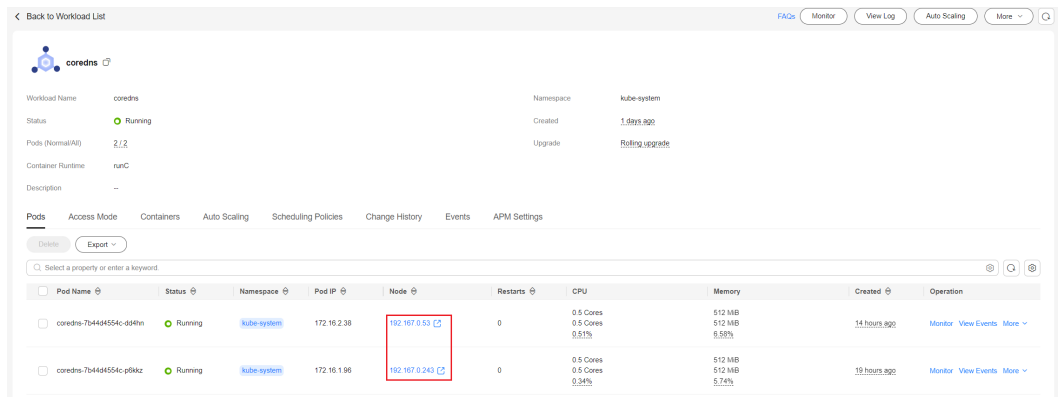
Step 5 Choose **Workload**, click the **Deployments** tab, and view the CoreDNS pods. Select the **kube-system** namespace to view the pod distribution of the add-on.

Figure 4-6 Viewing the deployment and distribution of CoreDNS pods



Step 6 View that the Deployment pods of the add-on has been allocated to nodes in two AZs.

Figure 4-7 Viewing CoreDNS pod distribution



----End

5 Security

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

Two methods are available:

- Method 1: Set the **automountServiceAccountToken** field of the service account to **false**. After the configuration is complete, newly created workloads

will not be associated with the default service account by default. Configure this field for each namespace as required.

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

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

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

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

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

Configuring Proper Cluster Access Permissions for Users

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

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

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.
- Cloud Native 2.0 network

In the Cloud Native Network 2.0 model, you can configure security groups to isolate networks between pods.

- 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-mode=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 Configuration Suggestions on CCE 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](#).



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 cluster as user **root** and run the following command:


```
iptables -I OUTPUT -s {container_cidr} -d 169.254.169.254 -j REJECT
```

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

To ensure configuration persistence, write the command to the **/etc/rc.local** script.
 - b. Run the following commands in the container to access the **userdata** and **metadata** interfaces of OpenStack and check whether the request is intercepted:


```
curl 169.254.169.254/openstack/latest/meta_data.json
curl 169.254.169.254/openstack/latest/user_data
```
- Container tunnel network
 - a. Log in to each node in the cluster as user **root** and run the following command:


```
iptables -I FORWARD -s {container_cidr} -d 169.254.169.254 -j REJECT
```

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

To ensure configuration persistence, write the command to the **/etc/rc.local** script.
 - b. Run the following commands in the container to access the **userdata** and **metadata** interfaces of OpenStack and check whether the request is intercepted:


```
curl 169.254.169.254/openstack/latest/meta_data.json
curl 169.254.169.254/openstack/latest/user_data
```
- CCE Turbo cluster

No additional configuration is required.

----End

5.3 Configuration Suggestions on CCE Container Runtime Security

Container technology uses Linux namespaces and cgroups to isolate and control resources between containers and nodes. Namespaces provide kernel-level

isolation, allowing processes to be restricted from accessing specific sets of resources, such as file systems, networks, processes, and users. Cgroups are a Linux kernel feature that manages and limits the usage of resources, such as CPU, memory, disk, and network, to prevent a single process from consuming too many resources and negatively impacting the overall system performance.

While namespaces and cgroups isolate resources between containers and nodes in an environment, node resources are not visible to containers. However, this isolation does not provide true security isolation because containers share the kernels of their nodes. If a container exhibits malicious behavior or a kernel vulnerability is exploited by attackers, the container may breach resource isolation. This can result in the container escaping and potentially compromising the node and other containers on the node.

To enhance runtime security, there are various mechanisms that can be used to detect and prevent malicious activities in containers. These mechanisms, such as capabilities, seccomp, AppArmor, and SELinux, can be integrated into Kubernetes. By using these mechanisms, container security can be improved and potential threats can be minimized.

Capabilities

Capabilities are a permission mechanism that enables a process to perform certain system operations without requiring full root permissions. This mechanism divides root permissions into smaller, independent permissions known as capabilities. By doing so, the process only obtains the minimum permission set necessary to complete its tasks. This approach enhances system security and helps mitigate potential security risks.

In a containerized environment, you can manage a container's capabilities by configuring its **securityContext**. The following is a configuration example:

```
...
securityContext:
  capabilities:
    add:
      - NET_BIND_SERVICE
    drop:
      -all
```

In this way, you can ensure that the container only has the necessary permissions to complete its tasks. This approach eliminates the risk of security breaches caused by excessive permissions. For more information about how to configure capabilities for a container, see [Set capabilities for a Container](#).

Seccomp

Seccomp is a mechanism that filters system calls, limiting the ones that processes can use to decrease the potential attack surface. Linux has many system calls, but not all are needed for containerized applications. By restricting the system calls that containers can execute, you can greatly reduce the risk of attacks on your applications.

Seccomp's main principle is to intercept all system calls and only allow trusted ones to pass. Container runtimes, such as Docker and containerd, come with default seccomp configurations that work for most common workloads.

In Kubernetes clusters, you can configure seccomp policies for containers to use the default security configuration. The following shows how to configure seccomp in different versions of Kubernetes clusters:

- For clusters of versions earlier than Kubernetes 1.19, you can use the following annotations to specify the seccomp configuration:

```
annotations:  
  seccomp.security.alpha.kubernetes.io/pod: "runtime/default"
```

- For clusters of Kubernetes 1.19 and later versions, you can use **securityContext** to configure seccomp policies.

```
securityContext:  
  seccompProfile:  
    type: RuntimeDefault
```

These configurations use the default seccomp policy, which permits containers to make a limited number of secure system calls. For more configuration options and advanced settings of seccomp, see [Restrict a Container's Syscalls with seccomp](#).

AppArmor and SELinux

AppArmor and SELinux are both Mandatory Access Control (MAC) systems that offer a more stringent approach than traditional Discretionary Access Control (DAC) to manage and restrict process permissions. While similar to seccomp in concept, these systems provide more precise access control, including access to file system paths, network ports, and other resources.

AppArmor and SELinux enable administrators to create policies that precisely manage the resources that applications can access. They can limit read and write permissions on specific files or directories, or regulate access to network ports.

Both systems are integrated into Kubernetes, allowing security policies to be applied at the container level.

- For details about how to use AppArmor, see [Restrict a Container's Access to Resources with AppArmor](#).
- For SELinux, you can configure **seLinuxOptions** in **securityContext**.

```
...  
securityContext:  
  seLinuxOptions:  
    level: "s0:c123,c456"
```

For details, see [Assign SELinux labels to a Container](#).

5.4 Configuration Suggestions on CCE 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.

To achieve strong isolation, like in logical multi-tenancy situations, it is important to have system add-ons run on separate nodes or node pools. This helps keep them separated from service pods and reduces the risk of privilege escalation within a cluster. To do this, you can set the node affinity policy to either **Node Affinity** or **Specified Node Pool Scheduling** on the add-on installation page.

Node Affinity

Not configured

Node Affinity

Specified Node Pool Scheduling

Custom

user can specify the node where the plug-in Deployment instance is deployed. (0 nodes selected) . If on the default cluster scheduling policy.

🔍 Select a property or enter a keyword.

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.

Networking Configuration

Network Model	VPC network
VPC	vpc-cce
Subnet	subnet-cce
Container CIDR Block	10.0.0.0/16 Add Container CIDR Block
Service CIDR Block	10.247.0.0/16
Forwarding	iptables
Default Node Security Group	cce-test-cce-node-99es4

Connection Information

Private IP	https://192.168.0.80:6443
EIP	-- Bind
Custom SAN	--
kubectl	Learn more
Certificate Authentication	X.509 certificate Download

Step 2 Configure access rules.

- CCE cluster: Log in to each node in the cluster as user **root** and run the following command:
 - VPC network:


```
iptables -I OUTPUT -s {container_cidr} -d {Private API server IP} -j REJECT
```
 - Container tunnel network:


```
iptables -I FORWARD -s {container_cidr} -d {Private API server IP} -j REJECT
```

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

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

- **Type:** Select **IPv4**.
 - **Protocol & Port:** Enter **5443** based on the port in the intranet API server address.
 - **Destination:** Select **IP address** and enter the IP address of the internal API server.
- e. Click **OK**.
- Step 3** Run the following command in the container to access kube-apiserver and check whether the request is intercepted:

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

----End

5.5 Configuration Suggestions on CCE Container Image Security

Container images are the primary defense against external attacks and are crucial for securing applications, systems, and the entire supply chain. If an image is insecure, it can become a vulnerability for attackers to exploit. This can lead to the container escaping to its node, allowing attackers to access sensitive data on the node or use it as a launching pad to gain control over the entire cluster or tenant account. This section describes some recommended configurations to mitigate such risks.

Minimizing a Container Image

To improve container image security, it is recommended that you remove any unnecessary binary files. When using an unknown image from Docker Hub, you are advised to review the image content with a tool like Dive. Dive provides layer-by-layer details of an image, helping to identify potential security risks. For details, see [Dive](#).

For improved security, it is recommended that you delete binary files with `setuid` and `setgid` permissions, because these can be exploited to elevate permissions. It is also wise to remove shell tools and applications that could be used maliciously, like `nc` and `curl`. To locate files with `setuid` and `setgid` bits, use the following command:

```
find / -perm /6000 -type f -exec ls -ld {} \;
```

To remove special permissions from the obtained files, add the following command to your container image:

```
RUN find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true
```

Using Multi-Stage Builds

Multi-stage builds are a great way to create container images efficiently, especially in the CI process. With multi-stage builds, you can perform lint checks on source code or static code analysis during the build process, providing quick feedback to developers. There is no need to wait for the entire build to finish.

Multi-stage builds offer significant security advantages by allowing developers to include only necessary components in container images, excluding build tools and other unnecessary binary files. This approach reduces the attack surface of images and improves overall security.

For more information about the concepts, best practices, and advantages of multi-stage builds, see the [Docker documentation](#). This will help you create streamlined and secure container images while optimizing development and deployment processes.

Using SWR

SWR provides easy, secure, reliable management of container images throughout their lifecycles, featuring image push, pull, and deletion.

SWR stands out for its precise permissions management, allowing administrators to customize access permissions for different users with read, edit, and manage levels. This ensures image security and compliance, meeting the needs of team collaboration.

Additionally, SWR offers automatic deployment capabilities. You can set a trigger to automatically deploy updated image versions. When a new image version is released, SWR automatically triggers the application that uses the image in CCE to update it, streamlining CI/CD.

To further enhance SWR's security and flexibility, fine-grained permissions control can be added to IAM users. For details about authorization management, see [User Permissions](#).

Scanning an Image Using SWR

With SWR, you can easily scan and secure your images with just a few clicks. Image scanning provides a thorough security check for your private images in repositories. It detects potential vulnerabilities and offers rectification suggestions.

Using an Image Signature and Configuring a Signature Verification Policy

Image signature verification is a security measure that confirms whether a container image has been tampered with after its creation. The image creator can sign the image content, and a user can verify the image's integrity and source by checking the signature.

This verification is crucial in maintaining container image security. By using image signature verification, organizations can guarantee the security and reliability of their containerized applications and safeguard them from potential security risks.

Adding the USER Instruction to a Dockerfile to Run Commands as a Non-root User

Properly configuring user permissions during container build and deployment can greatly enhance container security. This not only helps prevent potential malicious activities, but also aligns with the principle of least privilege (PoLP).

By setting the USER instruction in Dockerfiles, subsequent commands are executed as non-root users, which is a standard security practice.

- Limited permissions: Running a container as a non-root user can also mitigate potential security risks, because attackers cannot gain full control over the node even if the container is attacked.
- Restricted access: Non-root users typically have limited permissions, which restrict their access to and operation capabilities on node resources.

In addition to Dockerfiles, the **securityContext** field in **podSpec** of Kubernetes can be used to configure user and group IDs and enforce security policies during container deployment.

5.6 Configuration Suggestions on CCE 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 seen by running **ls -l** in the **/etc/secret-volume/** directory, but can be viewed by running **ls -al**.

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, see [Managing Service Accounts](#).

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 deploying applications in pods, you can configure requested resources and resource limits for the pods to prevent unlimited usage of resources during peak hours. However, after the upper limit is reached, an application error may occur. Pod scaling can effectively resolve this issue. If the resource usage on the node increases to a certain extent, newly added pods cannot be scheduled to this node. In this case, CCE will add nodes accordingly.

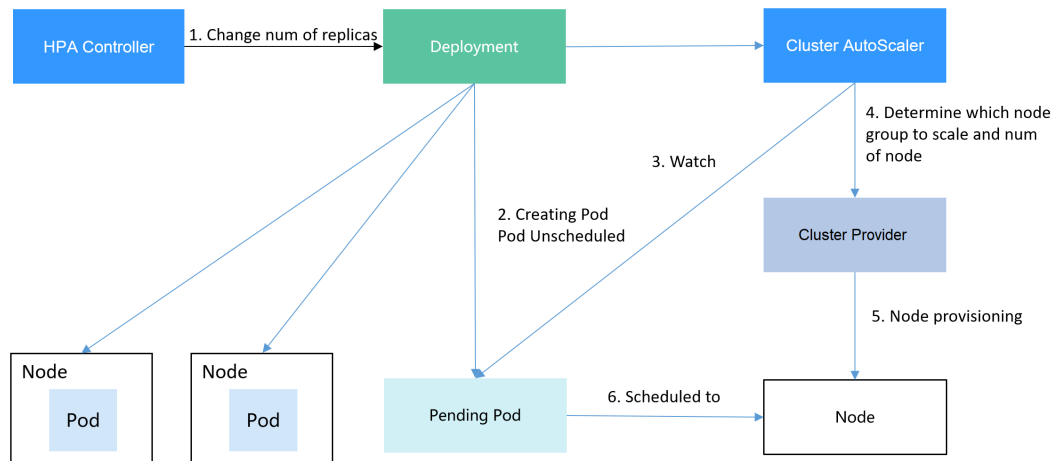
Solution

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 enables automatic scaling for most scenarios while also providing monitoring capabilities.

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
```

- Run the following command to build an image named **hpa-example** with the tag **latest**.

```
docker build -t hpa-example:latest .
```

- (Optional) Log in to the SWR console, choose **Organizations** in the navigation pane, and click **Create Organization** in the upper right corner. Skip this step if you already have an organization.
- 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.
- Run the login command copied in the previous step on the cluster node. If the login is successful, the message "Login Succeeded" is displayed.
- Tag the hpa-example image.

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

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

The following is an example:

```
docker tag hpa-example:latest {Image repository address}/group/hpa-example:latest
```

- 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 the node pool.

- Node Type**: Select a node type.

- **Specifications:** 2 vCPUs | 4 GiB

Retain the defaults for other parameters.

Step 3 Locate the row containing the newly created node pool and click **Auto Scaling** in the upper right corner.

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

- **Customize scale-out rules.:** Click **Add Rule**. In the dialog box displayed, configure parameters. If the CPU allocation rate is greater than 70%, a node is added to each associated node pool. A node scaling policy needs to be associated with a node pool. Multiple node pools can be associated. When you need to scale nodes, node with proper specifications will be added or reduced from the node pool based on the minimum waste principle.
- **Nodes:** Modify the node quantity range. The number of nodes in a node pool will always be within the range during auto scaling.
- **Cooldown Period:** a period during which the nodes added in the current node pool cannot be scaled in

Step 4 Click **OK**.

----End

Creating a Workload

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

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hpa-example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hpa-example
  template:
    metadata:
      labels:
        app: hpa-example
    spec:
      containers:
        - name: container-1
          image: 'hpa-example:latest' # Replace it with the address of the image you uploaded to SWR.
          resources:
            limits: # The value of limits must be the same as that of requests to prevent flapping
              during scaling.
              cpu: 500m
              memory: 200Mi
            requests:
              cpu: 500m
              memory: 200Mi
          imagePullSecrets:
            - name: default-secret
```

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

```
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 the pending state. During this period, nodes are being scaled out.

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

By using HPA and CA, auto scaling can be effortlessly implemented in various scenarios. Additionally, the scaling process of nodes and pods can be conveniently tracked.

7 Monitoring

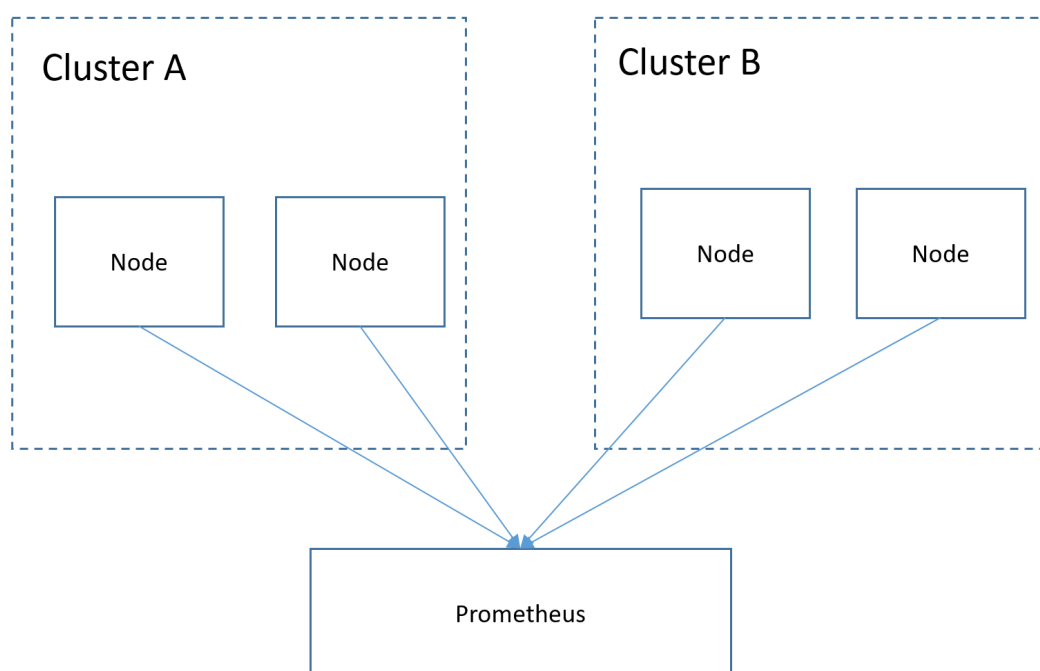
7.1 Monitoring Multiple Clusters Using Prometheus

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
```



```
[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]#
```

Tokens of the target clusters

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

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="k8s02" 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="k8s02" instance="192.168.0.162" job="k8s02_cAdvisor"	7.279s	65.193ms	

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="k8s" 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="k8s" 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="k8s" 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="k8s" instance="192.168.0.228" job="k8s_cAdvisor"	416.000ms	21.256ms	

The screenshot shows the Prometheus 'Targets' page with a search filter 'container_cpu_load_average_10s[namespace="default"]'. Below the search bar, there is a table of monitoring tasks. Two tasks are highlighted with red boxes and arrows:

- Monitoring task of cluster 2:** Points to a task with labels 'cluster="k8s02"' and 'instance="192.168.0.110"'. The task is in a 'Down' state.
- Monitoring task of cluster 1:** Points to a task with labels 'cluster="k8s"' and 'instance="192.168.0.109"'. The task is in a 'Down' state.

----End

7.2 Reporting Prometheus Monitoring Data to a Third-Party Monitoring Platform

Application Scenarios

The Cloud Native Cluster Monitoring add-on can report Prometheus metrics collected from clusters to a specified platform, for example, AOM or a third-party platform that supports Prometheus metrics. This section explains how to configure settings for Cloud Native Cluster Monitoring to send collected metrics to a third-party's Prometheus instance.

Step 1: Obtain the Data Reporting Address

Prometheus provides standard Remote Write APIs. You can enter the source address (Remote Write URL) in the Cloud Native Cluster Monitoring add-on for storing the locally collected monitoring data in a Prometheus instance remotely.

- If the Prometheus instance for receiving data is provided by a third-party vendor, view the Remote Write URL on the vendor's console.
- If the Prometheus instance for receiving data is an on-premises one, the Remote Write URL is **https:// {prometheus_addr} /api/v1/write**, where {prometheus_addr} indicates the IP address and port number for external access.

Step 2: Obtain the Authentication Mode

- For the third-party Prometheus instance, go to the vendor's console to view the token or account password used for authorized access.
- For the on-premises Prometheus instance, perform the following steps to obtain a token:
 - a. If this Prometheus instance is deployed in a Kubernetes cluster, view the token in the corresponding container. If this Prometheus instance is deployed on a VM, skip this step.

```
kubectl exec -ti -n monitoring prometheus-server-0 sh
```

Replace the variables in the command as needed:

- *monitoring*: indicates the namespace where a Prometheus pod is in.
- *prometheus-server-0*: indicates the name of a Prometheus pod.

- b. Check the location of the configuration file.

```
ps -aux | grep prometheus
```

Information similar to the following is displayed:

```
bb-5-15 ps -aux|grep promethe
      1  7.0  0.7 2405988 400016 ?        Ssl  10:41  0:10 /bin/prometheus --web.console.templates=/etc/prometheus/consoles --web.console.libraries=/etc/prometheus/console_libraries --storage.tsdb.retention.time=1d --config=/etc/prometheus/config_out/prometheus.yml --storage.tsdb.path=/prometheus --web.enable-lifecycle query-toolback delta-2x --enable-feature=remote-write-receiver --web.route-prefix/ --web.listen-address=:9090
bb-5-15 |
      27  0.0  0.0  3400  3000 pts/0    S+   10:52  0:00 grep promethe
```

- c. View and record the token information in **prometheus.env.yaml**.

```
cat /etc/prometheus/config_out/prometheus.env.yaml
```

```
alerting:
  alert_relabel_configs:
  - action: labeldrop
    regex: prometheus_replica
  alertmanagers:
  - path_prefix: /
    scheme: http
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - monitoring
  api_version: v2
  relabel_configs:
  - action: keep
    source_labels:
    - __meta_kubernetes_service_name
    regex: alertmanager
  - action: keep
    source_labels:
    - __meta_kubernetes_endpoint_port_name
    regex: http-web
remote_write:
  - url: https://100.79.29.98:8149/v1/7a34c707d93f4d91960567949ded4b50/b578a7c9-e17e-450c-ac8d-f69e88fa657b/push
    remote_timeout: 30s
  headers:
    cluster_id: 83f85f1d-6ef0-11ee-bacc-0255ac100b0e
    cluster_name: cce-00492128-v125
    name: cia_write_aom
  tls_config:
    insecure_skip_verify: true
  authorization:
    type: Bearer
    credentials: 0A0000001000000012A0CC6F5R1NDFE40907R9C9E46488CB198R0BE91033F01R62FD9461841CD40EF7313181FDAF670B5
```

Step 3: Connect to a Third-Party Monitoring Platform

- Step 1** Log in to the CCE console, click the name of a cluster with the Cloud Native Cluster Monitoring add-on installed to access the cluster console.

Step 2 In the navigation pane, choose **Settings** and click the **Monitoring** tab.

Step 3 Enable **Connect with third-party monitoring platforms** so that the data collected by Cloud Native Cluster Monitoring can be reported to a third-party monitoring platform.

- **Source Address:** Remote Write URL obtained in [step 1](#), for example, **https://127.0.0.1:9090/api/v1/write**.
- **Authentication method:** Select the authentication method supported by the third-party monitoring platform in [step 2](#).
 - **Basic Auth:** Enter the user name and password.
 - **Bearer Token:** Enter the identity credential (token).

Connect with third-party monitoring platforms ?

Profile data is reported to a third-party monitoring platform

Enabled

Source Address

https://127.0.0.1:9090/api/v1/write

Enter a complete RemoteWrite address, for example, https://127.0.0.1:9090/api/v1/write.

Authentication method

Basic Auth

Bearer Token

account

password

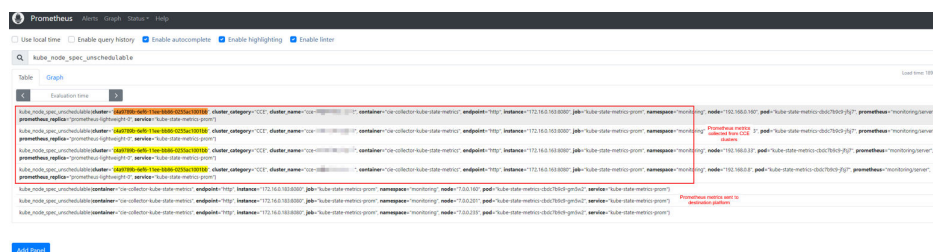
test

Step 4 Click **Confirm Configuration**.

----End

Step 4: Check the Data Sending and Receiving Statuses

After the preceding configuration is complete, log in to the Prometheus console supported by the third-party platform and view the Prometheus metrics with remote write on the **Graph** page.



8 Cluster

8.1 Suggestions on CCE Cluster Selection

When you use CCE to create a Kubernetes cluster, there are multiple configuration options and terms. This section compares the key configurations for CCE clusters and provides recommendations to help you create a cluster that better suits your needs.

Cluster Types

CCE supports CCE Turbo clusters and CCE standard clusters to meet your requirements. This section describes the differences between these two types of clusters.

Table 8-1 Cluster types

Category	Subcategory	CCE Turbo Cluster	CCE Standard Cluster
Cluster	Positioning	Next-gen container cluster designed for Cloud Native 2.0, with accelerated computing, networking, and scheduling	Standard cluster for common commercial use
	Node type	Deployment of VMs and bare-metal servers	Deployment of VMs and bare-metal servers
Networking	Model	Cloud Native Network 2.0: applies to large-scale and high-performance scenarios. Max networking scale: 2,000 nodes	Cloud Native Network 1.0: applies to common, smaller-scale scenarios. <ul style="list-style-type: none"> • Tunnel network model • VPC network model

Category	Subcategory	CCE Turbo Cluster	CCE Standard Cluster
	Performance	Flattens the VPC network and container network into one, achieving zero performance loss.	Overlays the VPC network with the container network, causing certain performance loss.
	Container network isolation	Associates pods with security groups. Unifies security isolation in and out the cluster via security groups' network policies.	<ul style="list-style-type: none"> • Tunnel network model: supports network policies for intra-cluster communications. • VPC network model: supports no isolation.
Security	Isolation	<ul style="list-style-type: none"> • Physical machine: runs Kata containers, allowing VM-level isolation. • VM: runs common containers, isolated by cgroups. 	Runs common containers, isolated by cgroups.

Cluster Versions

Due to the fast iteration, many bugs are fixed and new features are added in the new Kubernetes versions. The old versions will be gradually eliminated. When creating a cluster, select the latest commercial version supported by CCE.

Network Models

This section describes the network models supported by CCE clusters. You can select one model based on your requirements.

NOTICE

After clusters are created, the network models cannot be changed. Exercise caution when selecting the network models.

Table 8-2 Network model comparison

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Application scenarios	<ul style="list-style-type: none"> • Low requirements on performance: As the container tunnel network requires additional VXLAN tunnel encapsulation, it has about 5% to 15% of performance loss when compared with the other two container network models. Therefore, the container tunnel network applies to the scenarios that do not have high performance requirements, such as web applications, and middle-end and back-end services with a small number of access requests. • Large-scale networking: Different from the VPC network that is limited by the VPC route quota, the container tunnel network does not have any restriction 	<ul style="list-style-type: none"> • High performance requirements: As no tunnel encapsulation is required, the VPC network model delivers the performance close to that of a VPC network when compared with the container tunnel network model. Therefore, the VPC network model applies to scenarios that have high requirements on performance, such as AI computing and big data computing. • Small- and medium-scale networks: Due to the limitation on VPC routing tables, it is recommended that the number of nodes in a cluster be less than or equal to 1000. 	<ul style="list-style-type: none"> • High performance requirements: Cloud Native 2.0 networks use VPC networks to construct container networks, eliminating the need for tunnel encapsulation or NAT when containers communicate. This makes Cloud Native 2.0 networks ideal for scenarios that demand high bandwidth and low latency, such as live streaming and e-commerce flash sales. • Large-scale networking: Cloud Native 2.0 networks support a maximum of 2,000 ECS nodes and 100,000 pods.

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
	on the infrastructure. In addition, the container tunnel network controls the broadcast domain to the node level. The container tunnel network supports a maximum of 2000 nodes.		
Core technology	OVS	IPvlan and VPC route	VPC ENI/sub-ENI
Applicable clusters	CCE standard cluster	CCE standard cluster	CCE Turbo cluster
Container network isolation	Kubernetes native NetworkPolicy for pods	No	Pods support security group isolation.
Interconnecting pods to a load balancer	Interconnected through a NodePort	Interconnected through a NodePort	Directly interconnected using a dedicated load balancer Interconnected using a shared load balancer through a NodePort
Managing container IP addresses	<ul style="list-style-type: none"> Separate container CIDR blocks needed Container CIDR blocks divided by node and dynamically added after being allocated 	<ul style="list-style-type: none"> Separate container CIDR blocks needed Container CIDR blocks divided by node and statically allocated (the allocated CIDR blocks cannot be changed after a node is created) 	Container CIDR blocks divided from a VPC subnet (You do not need to configure separate container CIDR blocks.)

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Network performance	Performance loss due to VXLAN encapsulation	No tunnel encapsulation, and cross-node traffic forwarded through VPC routers (The performance is so good that is comparable to that of the host network, but there is a loss caused by NAT.)	Container network integrated with VPC network, eliminating performance loss
Networking scale	A maximum of 2000 nodes are supported.	Suitable for small- and medium-scale networks due to the limitation on VPC routing tables. It is recommended that the number of nodes be less than or equal to 1000. Each time a node is added to the cluster, a route is added to the VPC routing tables. Evaluate the cluster scale that is limited by the VPC routing tables before creating the cluster.	A maximum of 2000 nodes are supported. In a cloud-native network 2.0 cluster, containers' IP addresses are assigned from VPC CIDR blocks, and the number of containers supported is restricted by these blocks. Evaluate the cluster's scale limitations before creating it.

Cluster CIDR Blocks

There are node CIDR blocks, container CIDR blocks, and Service CIDR blocks in CCE clusters. When planning network addresses, note that:

- These three types of CIDR blocks cannot overlap with each other. Otherwise, a conflict will occur. All subnets (including those created from the secondary CIDR block) in the VPC where the cluster resides cannot conflict with the container and Service CIDR blocks.
- There are sufficient IP addresses in each CIDR block.
 - The IP addresses in a node CIDR block must match the cluster scale. Otherwise, nodes cannot be created due to insufficient IP addresses.
 - The IP addresses in a container CIDR block must match the service scale. Otherwise, pods cannot be created due to insufficient IP addresses.

In complex scenarios, for example, multiple clusters use the same VPC or clusters are interconnected across VPCs, determine the number of VPCs, the number of

subnets, the container CIDR blocks, and the communication modes of the Service CIDR blocks. For details, see [Planning CIDR Blocks for a Cluster](#).

Service Forwarding Modes

kube-proxy is a key component of a Kubernetes cluster. It is responsible for load balancing and forwarding between a Service and its backend pod.

CCE supports the iptables and IPVS forwarding modes.

- IPVS allows higher throughput and faster forwarding. It applies to scenarios where the cluster scale is large or the number of Services is large.
- iptables is the traditional kube-proxy mode. This mode applies to the scenario where the number of Services is small or there are a large number of short concurrent connections on the client.

If high stability is required and the number of Services is less than 2000, the iptables forwarding mode is recommended. In other scenarios, the IPVS forwarding mode is recommended.

Node Specifications

The minimum specifications of a node are 2 vCPUs and 4 GiB memory. Evaluate based on service requirements before configuring the nodes. However, using many low-specification ECSs is not the optimal choice. The reasons are as follows:

- The upper limit of network resources is low, which may result in a single-point bottleneck.
- Resources may be wasted. If each container running on a low-specification node needs a lot of resources, the node cannot run multiple containers and there may be idle resources in it.

Advantages of using large-specification nodes are as follows:

- The upper limit of the network bandwidth is high. This ensures higher resource utilization for high-bandwidth applications.
- Multiple containers can run on the same node, and the network latency between containers is low.
- The efficiency of pulling images is higher. This is because an image can be used by multiple containers on a node after being pulled once. Low-specifications ECSs cannot respond promptly because the images are pulled many times and it takes more time to scale these nodes.

Additionally, select a proper vCPU/memory ratio based on your requirements. For example, if a service container with large memory but fewer CPUs is used, configure the specifications with the vCPU/memory ratio of 1:4 for the node where the container resides to reduce resource waste.

Container Engines

CCE supports the containerd and Docker container engines. **containerd is recommended for its shorter traces, fewer components, higher stability, and less consumption of node resources.** Since Kubernetes 1.24, Dockershim is removed and Docker is no longer supported by default. For details, see [Kubernetes is Moving on From Dockershim: Commitments and Next Steps](#). CCE clusters do not support the Docker container engine.

Use containerd in typical scenarios. The Docker container engine is supported only in the following scenarios:

- Docker in Docker (usually in CI scenarios)
- Running the Docker commands on the nodes
- Calling Docker APIs

Node OS

Service container runtimes share the kernel and underlying calls of nodes. To ensure compatibility, select a Linux distribution version that is the same as or close to that of the final service container image for the node OS.

8.2 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. Huawei Cloud EulerOS 2.0 custom images are not supported.
 - 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 **/tmp/example.package** in the temporary server.

Step 2 Install Packer.

Download and install the [HashiCorp Packer](#). For details, see [Install Packer](#).

NOTE

The Packer version needs to be 1.10.0.

Take the CentOS 7 executor as an example. Run the following command to automatically install Packer (**This example is for reference only. For detailed operations, see the official guide**):

```
# Configure the yum repository and install Packer.
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/RHEL/hashicorp.repo
sudo yum -y install packer-1.10.0

# Configure an alias to avoid duplicate Packer binary in the OS and check the Packer version.
rpm -q packer
alias packer=$(rpm -ql packer)
packer -v
```

Step 3 Define Packer template parameters.

The `cce-image-builder/turbo-node.pkr.hcl` file defines the process of building an image using Packer. For details, see [Packer Documentation](#).

 NOTE

- Parameters of **variables** or **variable**
turbo-node.pkr.hcl defines the parameters required in the process of building an image. You can configure the parameters based on the live environment. For details, see [Table 1](#).
- Parameter of **packer**

required_plugins defines the add-on dependency of Packer, including the add-on source and version range. When you run **packer init**, the add-on is automatically downloaded and initialized.

```
packer {
  required_plugins {
    huaweicloud = {
      version = ">= 1.0.4"
      source = "github.com/huaweicloud/huaweicloud"
    }
  }
}
```

- Parameter of **source**
The preceding defined variables are referred to automatically configure the parameters required for creating an ECS.
- Parameter of **build**

The scripts are executed from top to bottom. Common modules such as the file upload module and script execution shell module are supported. The corresponding scripts and files are stored in the **user-scripts** and **user-packages** directories, respectively, in **cce-image-builder**.

Example:

```
build {
  sources = ["source.huaweicloud-ecs.builder"]

  # Example:
  provisioner "file" {
    source   = "<source file path>"
    destination = "<destination file path>"
  }

  provisioner "shell" {
    scripts = [
      "<source script file: step1.sh>",
      "<source script file: step2.sh>"
    ]
  }

  provisioner "shell" {
    inline = ["echo foo"]
  }
}
```

Step 4 Configure environment variables.

Configure the following environment variables on the executor:

```
export REGION_NAME=xxx
export IAM_ACCESS_KEY=xxx
export IAM_SECRET_KEY=xxx
export ECS_VPC_ID=xxx
export ECS_NETWORK_ID=xxx
export ECS_SECGRP_ID=xxx
export CCE_SOURCE_IMAGE_ID=xxx
export PKR_VAR_ecs_flavor=xxx
```

Table 8-3 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_SECGROUP_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.
PKR_VAR_ecs_flavor	Specifications of a temporary ECS	Enter a node flavor supported by CCE. The recommended flavor is 2 vCPUs and 4 GiB memory or higher.

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 **pkrc.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-shell1759174699
=> huaweicloud-ecs.builder: Setting a 15m0s timeout for the next provisioner...
=> huaweicloud-ecs.builder: Uploading packer_20210530105050.log -> /var/log/packer_20210530105050.log
=> huaweicloud-ecs.builder: packer_20210530105050.log 43.63 KiB / 43.58 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-20210530105050
=> huaweicloud-ecs.builder: Waiting for image image-by-packer-20210530105050 to become available ...
=> huaweicloud-ecs.builder: Image: 64e940f4-d674-4ae1-89cc-299501581c59
=> huaweicloud-ecs.builder: Deleted temporary floating IP '494617cc-a7e9-442a-b3e8-3b98c2c3f804' (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

8.3 Connecting to Multiple Clusters Using kubectl

Background

The kubectl command line tool relies on the kubeconfig configuration file to locate the necessary authentication information to select a cluster and communicate with its API server. By default, kubectl uses the **\$HOME/.kube/config** file as the credential for accessing the cluster.

When working with CCE clusters on a daily basis, it is common to manage multiple clusters simultaneously. However, this can make using the kubectl command line tool to connect to clusters cumbersome, as it requires frequent switching of the kubeconfig file during routine O&M. This section introduces how to connect to multiple clusters using the same kubectl client.

NOTE

The file used to configure cluster access is called the kubeconfig file, but it does not mean that the file name is **kubeconfig**.

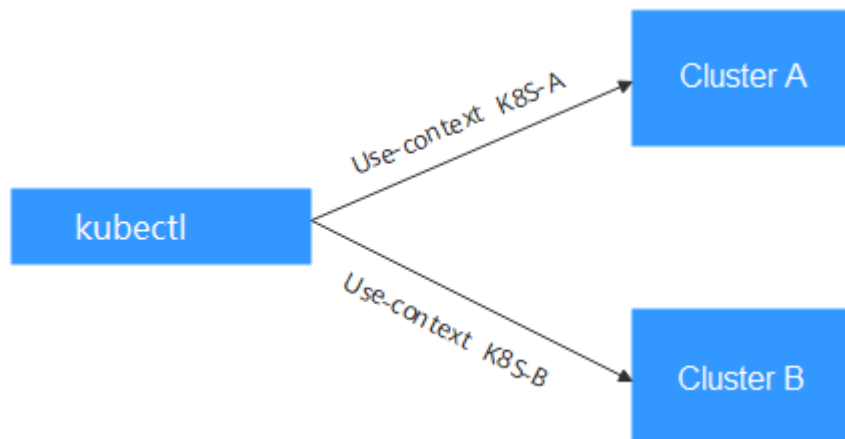
Solution

When performing O&M on Kubernetes clusters, it is often necessary to switch between multiple clusters. The following shows some typical solutions for cluster switchover:

- **Solution 1:** Specify **--kubeconfig** of kubectl to select the kubeconfig file used by each cluster and use aliases to simplify commands.
- **Solution 2:** Combine clusters, users, and credentials in multiple kubeconfig files into one configuration file and run **kubectl config use-context** to switch clusters.

Compared with solution 1, this solution requires manual configuration of the kubeconfig file, which is relatively complex.

Figure 8-1 Using kubectl to connect to multiple clusters



Prerequisites

- You have a Linux VM with the kubectl command line tool installed. The kubectl version must match the cluster version. For details, see [Install Tools](#).
- The VM where kubectl is installed must be able to access the network of each cluster.

kubeconfig File Structure

kubeconfig is the configuration file of kubectl. You can download it on the cluster details page.

The content of the kubeconfig file 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": "LS0tLS1CRUdJTjBDRVJ...",
      "client-key-data": "LS0tLS1CRUdJTjBS..."
    }
  }
],
  "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 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.

Solution 1: Specify Different kubeconfig Files in Commands

Step 1 Log in to the VM where kubectl is installed.

Step 2 Download the kubeconfig files of the two clusters to the **/home** directory on the kubectl client. The following names are taken as examples.

Cluster Name	kubeconfig File Name
Cluster A	kubeconfig-a.json
Cluster B	kubeconfig-b.json

Step 3 Make kubectl access cluster A by default and move the **kubeconfig-a.json** file to **\$HOME/.kube/config**.

```

cd /home
mkdir -p $HOME/.kube
mv -f kubeconfig-a.json $HOME/.kube/config

```

Step 4 Move the **kubeconfig-b.json** file of cluster B to **\$HOME/.kube/config-test**.

```

mv -f kubeconfig-b.json $HOME/.kube/config-test

```

The name of the **config-test** file can be customized.

Step 5 Add **--kubeconfig** to specify the credential used by the kubectl commands when accessing cluster B. (There is no need to add **--kubeconfig** when running kubectl commands to access cluster A, because kubectl can access cluster A by default.) For example, run the following command to check the nodes in cluster B:

```

kubectl --kubeconfig=$HOME/.kube/config-test get node

```

If you frequently use a long command, the preceding method can be inconvenient. To simplify the command, you can use aliases. For example:

```

alias ka='kubectl --kubeconfig=$HOME/.kube/config'
alias kb='kubectl --kubeconfig=$HOME/.kube/config-test'

```


In the preceding information, **ka** and **kb** can be custom aliases. When running the `kubectl` command, you can directly enter **ka** or **kb** to replace **kubectl**. The `--kubeconfig` parameter is automatically added. For example, the command for checking nodes in cluster B can be simplified as follows:

```
kb get node
```

----End

Solution 2: Combine the kubeconfig Files of the Two Clusters Together

The following steps walk you through the procedure of modifying the kubeconfig files and accessing multiple clusters.

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 the kubeconfig files 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": "LS0tLS1CRUdJTzM...",
      "client-key-data": "LS0tLS1CRUdJTjB..."
    }
  }
],
  "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": "LS0rTUideUdJTjB...."
  }
}],
"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": "LS0tLS1CRUdJTjB...."
    }
  }
],
  "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": "LS0rTUideUdJTjB...."
    }
  }
],
  "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": "LS0tLS1CRUdJTzM...",
      "client-key-data": "LS0tLS1CRUdJTjB..."
    }
  },
  {
    "name": "Cluster-B-user",
    "user": {
      "client-certificate-data": "LS0tLS1CRUdJTzM...",
      "client-key-data": "LS0rTUideUdJTjB..."
    }
  }
],
  "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"
}
```

Step 4 Run the following command to copy the combined file to the kubectl configuration path:

```
mkdir -p $HOME/.kube
```

```
mv -f kubeconfig.json $HOME/.kube/config
```

Step 5 Run the `kubectl` command to check whether the two clusters can be accessed.

```
# 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/coresdns/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/coresdns/dns/proxy
```

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

If you frequently use a long command, the preceding method can be inconvenient. To simplify the command, you can use aliases. For example:

```
alias ka='kubectl config use-context Cluster-A-Context;kubectl'
alias kb='kubectl config use-context Cluster-B-Context;kubectl'
```

In the preceding information, **ka** and **kb** can be custom aliases. When running the `kubectl` command, you can directly enter **ka** or **kb** to replace **kubectl**. You need to switch the context and then run the `kubectl` command. For example:

```
# ka cluster-info
Switched to context "Cluster-A-Context".
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/coresdns/dns/proxy
```

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

----End

8.4 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 adjust the disk capacity to the minimum of 20 GiB or reduce the disk capacity attached to a node to the minimum of 10 GiB.

NOTICE

Adjusting the size of the data disk used by the container runtime and kubelet may incur risks. You are advised to evaluate the capacity adjustment and then perform the operations described in this section.

- If the disk capacity is too small, the image pull may fail. If different images need to be frequently pulled on the node, you are not advised to reduce the data disk capacity.
- Before a cluster upgrade, the system checks whether the data disk usage exceeds 95%. If the usage is high, the cluster upgrade may be affected.
- If Device Mapper is used, the disk capacity may be insufficient. You are advised to use the OverlayFS or select a large-capacity data disk.
- For dumping logs, application logs must be stored in a separate disk to prevent insufficient storage capacity of the dockersys volume from affecting service running.
- After reducing the data disk capacity, you are advised to install the npd add-on in the cluster to detect disk usage. If the disk usage of a node is high, resolve this problem by referring to [What If the Data Disk Capacity Is Insufficient?](#)

Constraints

- Only clusters of v1.19 or later allow reducing the capacity of the data disk used by container runtimes and kubelet.
- Only the EVS disk capacity can be adjusted. (Local disks are available only when the node specification is **disk-intensive** or **Ultra-high I/O**.)

Selecting a Data Disk

When selecting a data disk, consider the following factors:

- During image pull, the system downloads the image package (the .tar package) from the image repository, and decompresses the package. Then it deletes the package but retain the image file. During the decompression of the .tar package, the package and the decompressed image file coexist. Reserve the capacity for the decompressed files.
- Mandatory add-ons (such as everest and coredns) may be deployed on nodes during cluster creation. When calculating the data disk size, reserve about 2 GiB storage capacity for them.
- Logs are generated during application running. To ensure stable application running, reserve about 1 GiB storage capacity for each pod.

For details about the calculation formulas, see [OverlayFS](#) and [Device Mapper](#).

OverlayFS

By default, the container engine and container image storage capacity of a node using the OverlayFS storage driver occupies 90% of the data disk capacity (you are advised to retain this value). All the 90% storage capacity is used for dockersys partitioning. The calculation methods are as follows:

- Capacity for storing container engines and container images requires 90% of the data disk capacity by default.
 - Capacity for dockersys volume (in the `/var/lib/docker` directory) requires 90% of the data disk capacity. The entire container engine and container image capacity (need 90% of the data disk capacity by default) are in the `/var/lib/docker` directory.
- Capacity for storing temporary kubelet and emptyDir requires 10% of the data disk capacity.

On a node using the OverlayFS, when an image is pulled, the .tar package is decompressed after being downloaded. During this process, the .tar package and the decompressed image file are stored in the dockersys volume, occupying about twice the actual image storage capacity. After the decompression is complete, the .tar package is deleted. Therefore, during image pull, after deducting the storage capacity occupied by the system add-on images, ensure that the remaining capacity of the dockersys volume is greater than twice the actual image storage capacity. To ensure that the containers can run stably, reserve certain capacity in the dockersys volume for container logs and other related files.

When selecting a data disk, consider the following formula:

Capacity of dockersys volume > Actual total image storage capacity x 2 + Total system add-on image storage capacity (about 2 GiB) + Number of containers x Available storage capacity for a single container (about 1 GiB log storage capacity for each container)

NOTE

If container logs are output in the `json.log` format, they will occupy some capacity in the dockersys volume. If container logs are stored on persistent storage, they will not occupy capacity in the dockersys volume. Estimate the capacity of every container as required.

Example:

Assume that the node uses the OverlayFS and the data disk attached to this node is 20 GiB. According to [the preceding methods](#), the capacity for storing container engines and images occupies 90% of the data disk capacity, and the capacity for the dockersys volume is 18 GiB (20 GiB x 90%). Additionally, mandatory add-ons may occupy about 2 GiB storage capacity during cluster creation. If you deploy a .tar package of 10 GiB, the package decompression takes 20 GiB of the dockersys volume's storage capacity. This, coupled with the storage capacity occupied by mandatory add-ons, exceeds the remaining capacity of the dockersys volume. As a result, the image pull may fail.

Device Mapper

By default, the capacity for storing container engines and container images of a node using the Device Mapper storage driver occupies 90% of the data disk capacity (you are advised to retain this value). The occupied capacity includes the dockersys volume and thinpool volume. The calculation methods are as follows:

- Capacity for storing container engines and container images requires 90% of the data disk capacity by default.
 - Capacity for the dockersys volume (in the `/var/lib/docker` directory) requires 20% of the capacity for storing container engines and container images.

- Capacity for the 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 dockersys partition space and thinpool space 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 a data disk on the EVS console.

Only the storage capacity of the EVS disk is expanded. You also need to perform the following steps to expand the capacity of the logical volume and file system.

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 **dockersys**.

1. Check the disk and partition sizes of the device.

```
# 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 150G  0 disk  # The data disk has been expanded to 150 GiB, but 50 GiB
space is not allocated.
└─vgpaas-dockersys 253:0  0  90G  0 lvm  /var/lib/containerd
   └─vgpaas-kubernetes 253:1  0  10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

2. Expand the disk capacity.

Add the new disk capacity to the **dockersys** logical volume used by the container engine.

- a. Expand the PV capacity so that LVM can identify the new EVS capacity. `/dev/sdb` specifies the physical volume where dockersys is located.

```
pvresize /dev/sdb
```

Information similar to the following is displayed:

```
Physical volume "/dev/sdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

- b. Expand 100% of the free capacity to the logical volume. `vgpaas/dockersys` specifies the logical volume used by the container engine.

```
lvextend -l+100%FREE -n vgpaas/dockersys
```

Information similar to the following is displayed:

```
Size of logical volume vgpaas/dockersys changed from <90.00 GiB (23039 extents) to 140.00
GiB (35840 extents).
Logical volume vgpaas/dockersys successfully resized.
```

- c. Adjust the size of the file system. `/dev/vgpaas/dockersys` specifies the file system path of the container engine.

```
resize2fs /dev/vgpaas/dockersys
```

Information similar to the following is displayed:


```
Filesystem at /dev/vgpaas/dockersys is mounted on /var/lib/containerd; on-line resizing required
old_desc_blocks = 12, new_desc_blocks = 18
The filesystem on /dev/vgpaas/dockersys is now 36700160 blocks long.
```

3. Check whether the capacity is expanded.

```
# lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 50G 0 disk
├─sda1 8:1 0 50G 0 part /
└─vgpaas-dockersys 253:0 0 140G 0 lvm /var/lib/containerd
   └─vgpaas-kubernetes 253:1 0 10G 0 lvm /mnt/paas/kubernetes/kubelet
```

Devicemapper: A thin pool is allocated to store image data.

1. Check the disk and partition sizes of the device.

```
# lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda 8:0 0 50G 0 disk
├─vda1 8:1 0 50G 0 part /
└─vgdb 8:16 0 200G 0 disk
   └─vgpaas-dockersys 253:0 0 18G 0 lvm /var/lib/docker
      └─vgpaas-thinpool_tmeta 253:1 0 3G 0 lvm
         └─vgpaas-thinpool 253:3 0 67G 0 lvm # Space used by thinpool
            ...
      └─vgpaas-thinpool_tdata 253:2 0 67G 0 lvm
         └─vgpaas-thinpool 253:3 0 67G 0 lvm
            ...
   └─vgpaas-kubernetes 253:4 0 10G 0 lvm /mnt/paas/kubernetes/kubelet
```

2. Expand the disk capacity.

Option 1: Add the new disk capacity to the thin pool disk.

- a. Expand the PV capacity so that LVM can identify the new EVS capacity. `/dev/vdb` specifies the physical volume where thinpool is located.
`pvresize /dev/vdb`

Information similar to the following is displayed:

```
Physical volume "/dev/vdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

- b. Expand 100% of the free capacity to the logical volume. `vgpaas/thinpool` specifies the logical volume used by the container engine.
`lvextend -l+100%FREE -n vgpaas/thinpool`

Information similar to the following is displayed:

```
Size of logical volume vgpaas/thinpool changed from <67.00 GiB (23039 extents) to <167.00 GiB (48639 extents).
Logical volume vgpaas/thinpool successfully resized.
```

- c. Do not need to adjust the size of the file system, because the thin pool is not mounted to any devices.
- d. Check whether the capacity is expanded. Run the `lsblk` command to check the disk and partition sizes of the device. If the new disk capacity has been added to the thin pool, the capacity is expanded.

```
# lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
vda 8:0 0 50G 0 disk
├─vda1 8:1 0 50G 0 part /
└─vgdb 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 167G 0 lvm # Thin pool space after
            capacity expansion
            ...
      └─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
```

Option 2: Add the new disk capacity to the **dockersys** disk.

- a. Expand the PV capacity so that LVM can identify the new EVS capacity. */dev/vdb* specifies the physical volume where dockersys is located.

```
pvresize /dev/vdb
```

Information similar to the following is displayed:

```
Physical volume "/dev/vdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

- b. Expand 100% of the free capacity to the logical volume. *vgpaas/dockersys* specifies the logical volume used by the container engine.

```
lvextend -l+100%FREE -n vgpaas/dockersys
```

Information similar to the following is displayed:

```
Size of logical volume vgpaas/dockersys changed from <18.00 GiB (4607 extents) to <118.00 GiB (30208 extents).
Logical volume vgpaas/dockersys successfully resized.
```

- c. Adjust the size of the file system. */dev/vgpaas/dockersys* specifies the file system path of the container engine.

```
resize2fs /dev/vgpaas/dockersys
```

Information similar to the following is displayed:

```
Filesystem at /dev/vgpaas/dockersys is mounted on /var/lib/docker; on-line resizing required
old_desc_blocks = 3, new_desc_blocks = 15
The filesystem on /dev/vgpaas/dockersys is now 30932992 blocks long.
```

- d. Check whether the capacity is expanded. Run the **lsblk** command to check the disk and partition sizes of the device. If the new disk capacity has been added to the dockersys, the capacity is expanded.

```
# lsblk
NAME                                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                                  8:0    0  50G  0 disk
└─vda1                               8:1    0  50G  0 part /
vgdb                                 8:16    0 200G  0 disk
├─vgpaas-dockersys                  253:0    0 118G  0 lvm  /var/lib/docker # dockersys after
capacity expansion
├─vgpaas-thinpool_tmeta              253:1    0   3G  0 lvm
├─vgpaas-thinpool                    253:3    0  67G  0 lvm
├─...
├─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
```

----End

8.5 Protecting a CCE Cluster Against Overload

As services grow, the Kubernetes cluster scales up, putting more pressure on the control plane. If the control plane cannot handle the load, clusters may fail to provide services. This document explains the symptoms, impact, and causes of cluster overload, as well as how CCE clusters can protect against overload. It also provides recommended measures for protecting against overload.

What Is Cluster Overload?

An overloaded cluster can cause delays in Kubernetes API responses and increase the resource usage on master nodes. In severe cases, the APIs may fail to respond, master nodes may become unusable, and the entire cluster may malfunction.

When a cluster is overloaded, both the control plane and the services that rely on it are impacted. The following lists some scenarios that may be affected:

- Kubernetes resource management: Creating, deleting, updating, or obtaining Kubernetes resources may fail.
- Kubernetes distributed leader selection: In distributed applications based on Kubernetes Leases, leaders may restart due to lease renewal request timeout.

NOTE

For example, if the lease renewal of the controller component of the NPD add-on fails, an active/standby switchover is triggered. This means that the active instance will restart, and the standby instance will take over services, ensuring that there is no impact on services.

- Cluster management: When a cluster is severely overloaded, it may become unavailable. In this case, cluster management operations, such as creating or deleting nodes, cannot be performed.

Common causes of cluster overload are as follows:

- The cluster resource data volume is too large.
etcd and kube-apiserver are two core components of the cluster control plane. etcd serves as the background database that stores all cluster data, while kube-apiserver acts as the entry point for processing requests. kube-apiserver caches cluster data to lessen the burden on etcd, and other core components in the cluster also cache various resources and monitor changes to these resources.

However, if the cluster resource data volume is too large, the control plane resource usage remains high, leading to overload when the resource data volume exceeds the bearing capability.

- A large amount of data is obtained from a client. For example, a large number of LIST requests are initiated or a single LIST request is sent to obtain a large amount of data.

Assume that a client uses field selectors to obtain pod data in a cluster and needs to obtain data from etcd (although the client can also get data from the kube-apiserver cache). Data in etcd cannot be obtained by field, so kube-apiserver must get all pod data from etcd, replicate, and serialize structured pod data, and then respond to the client request.

When the client sends a LIST request, it may need to be processed by multiple control plane components, resulting in a larger amount of data to be processed and a more complex data type. As a result, when the client gets a large amount of data, resource usages on etcd and API server remain high. If the bearing capability is exceeded, the cluster becomes overloaded.

CCE Overload Control

- **Overload control:** CCE clusters have supported overload control since v1.23, which reduces the number of LIST requests outside the system when the

control plane experiences high resource usage pressure. To use this function, enable overload control for your clusters. For details, see [Enabling Overload Control for a Cluster](#).

- **Optimized processes on LIST requests:** Starting from CCE clusters of v1.23.8-r0 and v1.25.3-r0, processes on LIST requests have been optimized. Even if a client does not specify the **resourceVersion** parameter, kube-apiserver responds to requests based on its cache to avoid additional etcd queries and ensure that the response data is up to date. Additionally, namespace indexes are now added to the kube-apiserver cache. This means that when a client requests a specified resource in a specified namespace, it no longer needs to obtain resources belonging to the namespace based on full data. This effectively reduces the response delay and control plane memory overhead.
- **Refined traffic limiting policy on the server:** The API Priority and Fairness (APF) feature is used to implement fine-grained control on concurrent requests. For details, see [API Priority and Fairness](#).

Suggestions

This section describes measures and suggestions you can take to prevent clusters from being overloaded.

Upgrading the Cluster Version

As the CCE cluster version evolves, new overload protection features and optimizations are regularly introduced. It is recommended that you promptly upgrade your clusters to the latest version. For details, see [Upgrading a Cluster](#).

Enabling Overload Control

After overload control is enabled, concurrent LIST requests outside the system will be dynamically controlled based on the resource demands received by master nodes to ensure the stable running of the master nodes and the cluster.

For details, see [Cluster Overload Control](#).

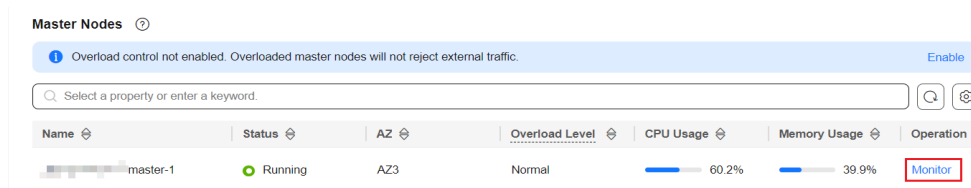
Enabling Observability

Observability is crucial for maintaining the reliability and stability of clusters. By using monitoring, alarms, and logs, administrators can gain a better understanding of the clusters' performance, promptly identify any issues, and take corrective action in a timely manner.

Monitoring configurations

- You can check the monitoring information about master nodes on the **Overview** page of the CCE cluster console.

Figure 8-2 Viewing master node monitoring information



- You can also use Prometheus to monitor the metrics of master node components, especially the memory usage, resource quantity, QPS, and request latency of kube-apiserver. For details, see [Monitoring Metrics of Master Node Components Using Prometheus](#).

Controlling Data Volume of Resources

When the resource data volume in a cluster is too large, it can negatively impact etcd performance, including data read and write latency. Additionally, if the data volume of a single type of resource is too large, the control plane consumes a significant number of resources when a client requests all the resources. To avoid these issues, it is recommended that you keep both the etcd data volume and the data volume of a single type of resources under control.

Table 8-4 Recommended maximum etcd data volume for different cluster scales

Cluster Scale	50 Nodes	200 Nodes	1,000 Nodes	2,000 Nodes
Total etcd data capacity	500Mi	1Gi	4Gi	8Gi
etcd data volume of a single type of resources	50Mi	100Mi	400Mi	800Mi

Clearing Unused Resources

To prevent a large number of pending pods from consuming extra resources on the control plane, it is recommended that you promptly clear up Kubernetes resources that are no longer in use, such as ConfigMaps, Secrets, and PVCs.

Optimizing the Client Access Mode

- To avoid frequent LIST queries, it is best to use the client cache mechanism when retrieving cluster resource data multiple times. It is recommended that you communicate with clusters using informers and listers. For details, see [client-go documentation](#).

If a LIST query must be used, you can:

- Obtain needed data from the kube-apiserver cache first and avoid making additional queries on etcd data. For clusters earlier than v1.23.8-r0 and v1.25.3-r0, you can set **resourceVersion** to **0**. In clusters of v1.23.8-r0, v1.25.3-r0, and later versions, CCE has improved the way data is retrieved and ensured that the cached data is up to date. By default, you can access the required data from the cache.
- Accurately define the query scope to avoid retrieving irrelevant data and using unnecessary resources. For example:

```
# client-go Code example for obtaining pods in a specified namespace
k8sClient.CoreV1().Pods("<your-namespace>").List(metav1.ListOptions{})
# kubectl Command example for obtaining pods in a specified namespace
kubectl get pods -n <your-namespace>
```

- Use the more efficient Protobuf format instead of the JSON format. By default, Kubernetes returns objects serialized to JSON with content type **application/json**. This is the default serialization format for the API. However, clients may request the more efficient Protobuf representation of these objects for better performance. For details, see [Alternate representations of resources](#).

Changing the Cluster Scale

If the resource usage on the master nodes in a cluster remains high for a long time, for example, the memory usage is greater than 85%, it is recommended that you promptly increase the cluster management scale. This will prevent the cluster from becoming overloaded during sudden traffic surges. For details, see [Changing Cluster Scale](#).

NOTE

- The performance of the master nodes improves and their specifications become higher as the management scale of a cluster increases.
- The CCE cluster management scale is the maximum number of nodes that a cluster can manage. It is used as a reference during service deployment planning, and the actual quantity of nodes in use may not reach the maximum number of nodes selected. The actual scale depends on various factors, including the type, quantity, and size of resource objects in the cluster, as well as the number of external accesses to the cluster control plane.

Splitting the Cluster

The Kubernetes architecture has a performance bottleneck, meaning that the scale of a single cluster cannot be expanded indefinitely. If your cluster has 2,000 worker nodes, it is necessary to split the services and deploy them across multiple clusters. If you encounter any issues with splitting a cluster, submit a service ticket for technical support.

Summary

When running services on Kubernetes clusters, their performance and availability are influenced by various factors, including the cluster scale, number and size of resources, and resource access. CCE has optimized cluster performance and availability based on cloud native practices and has developed measures to protect against cluster overload. You can use these measures to ensure that your services run stably and reliably over the long term.

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

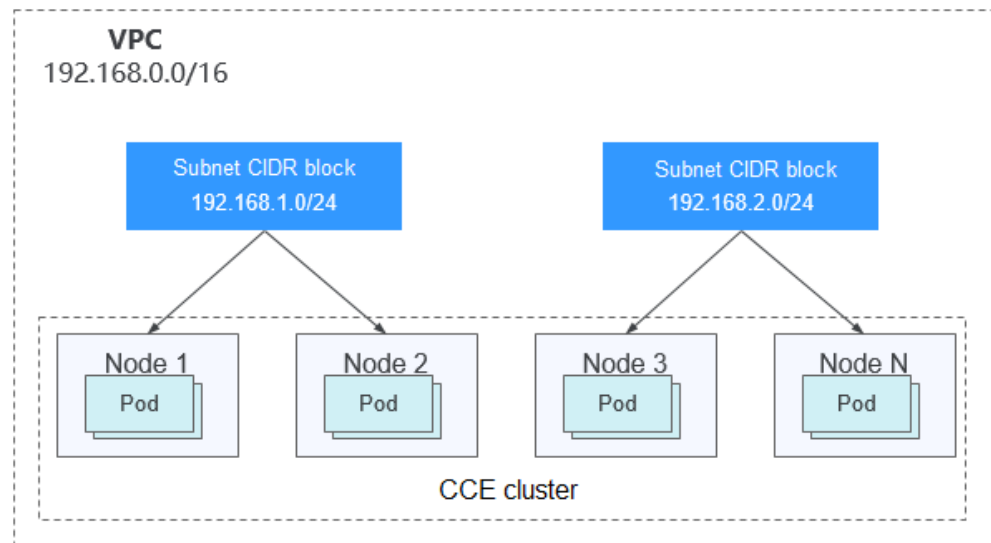
- **VPC CIDR Block**

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**

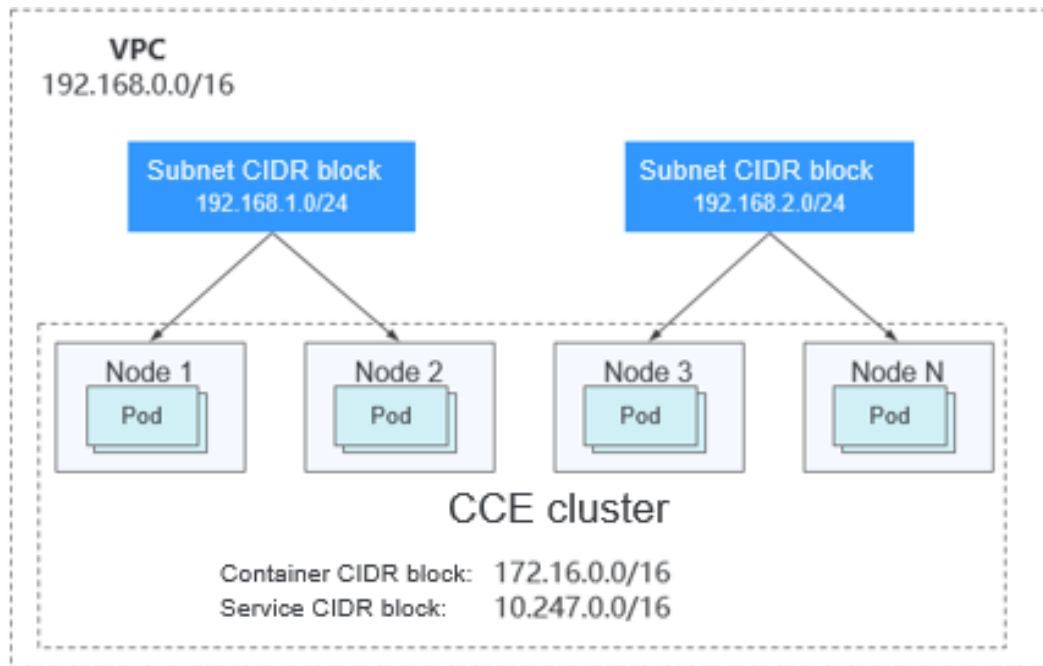
Service is also a Kubernetes concept. Each Service has an address. When creating a cluster on CCE, you can specify the Service CIDR block. Similarly, the Service CIDR block cannot overlap with the subnet CIDR block or the container CIDR block. The Service CIDR block can be used only within a cluster.

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)



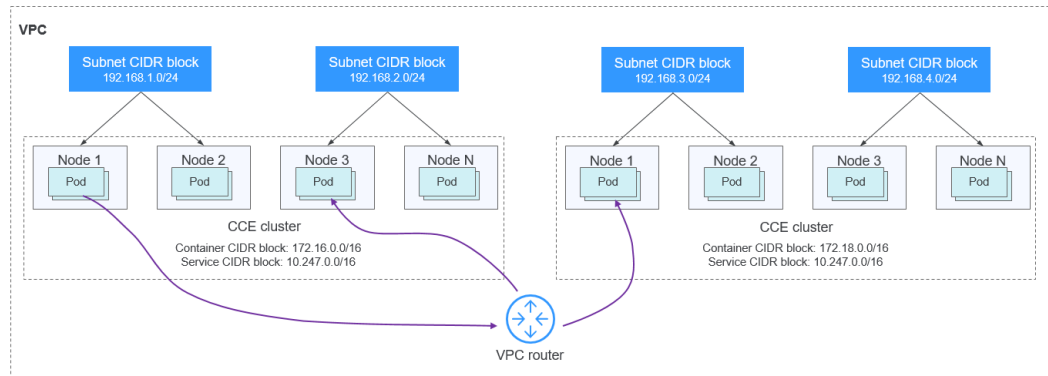
Single-VPC Multi-Cluster Scenarios

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, if the node security group allows container CIDR block from the peer cluster, pods in one cluster can directly access pods in another cluster through the pod IP addresses.
- Service CIDR Block: can be used only in clusters. Therefore, the Service CIDR blocks of different clusters can overlap, but cannot overlap with the subnet CIDR block and container CIDR block of the cluster.

Figure 9-3 VPC network - multi-cluster scenario

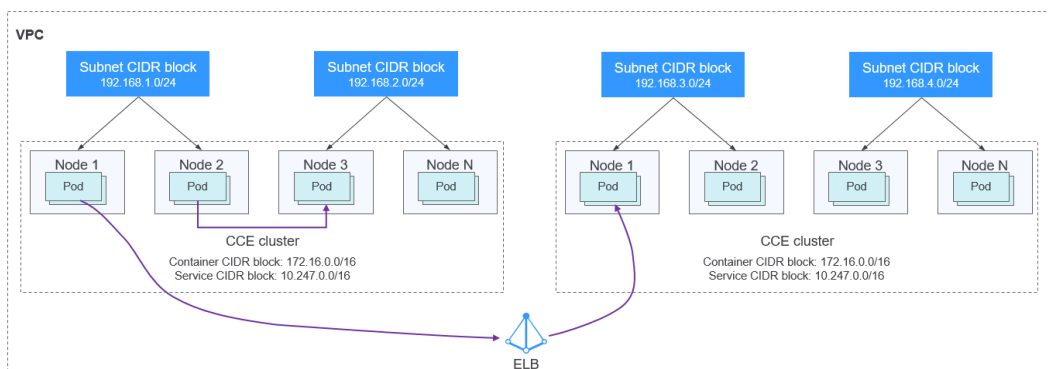


Tunnel network model

Though at some cost of performance, the tunnel encapsulation enables higher interoperability and compatibility with advanced features (such as network policy-based isolation), meeting the requirements of most applications. **Figure 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 through pod IP addresses. Services are needed for accessing pods in different clusters. The LoadBlancer Services are recommended.
- Service CIDR Block: can be used only in clusters. Therefore, the Service CIDR blocks of different clusters can overlap, but cannot overlap with the subnet CIDR block and container CIDR block of the cluster.

Figure 9-4 Tunnel network - multi-cluster scenario



Clusters using different networks

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

- **VPC CIDR Block:** In this scenario, all clusters are located in the same VPC CIDR block. Ensure that there are sufficient available IP addresses in the VPC.
- **Subnet CIDR Block:** Ensure that the subnet CIDR block does not overlap with the container CIDR block.
- **Container CIDR Block:** Ensure that the container CIDR blocks of clusters in **VPC network model** do not overlap.
- **Service CIDR Block:** The Service CIDR blocks of all clusters can overlap, but cannot overlap with the subnet CIDR block and container CIDR block of the cluster.

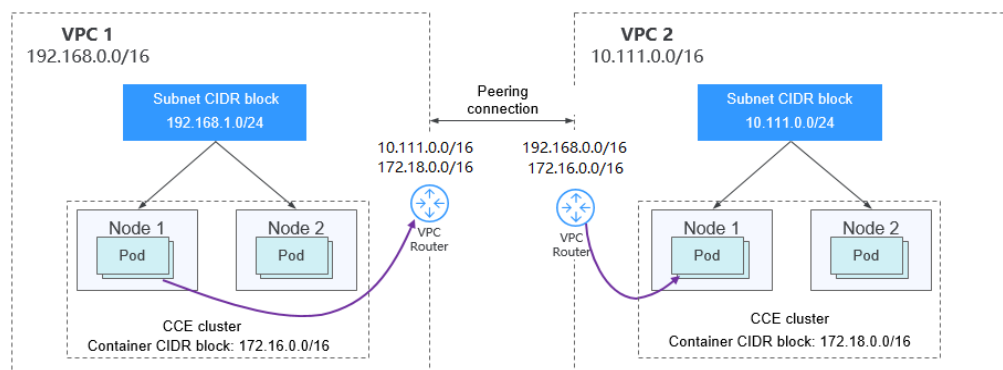
Cross-VPC Cluster Interconnection

If VPCs cannot communicate with each other, a VPC peering connection is used to ensure communication between VPCs. When two VPC networks are interconnected, you can configure the packets to be sent to the peer VPC in the route table. For details, see [VPC Peering Connection Overview](#).

Clusters using VPC networks

To allow clusters that use VPC networks to access each other across VPCs, add routes to the two ends of the VPC peering after a VPC peering connection is created.

Figure 9-5 VPC network - VPC interconnection scenario



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

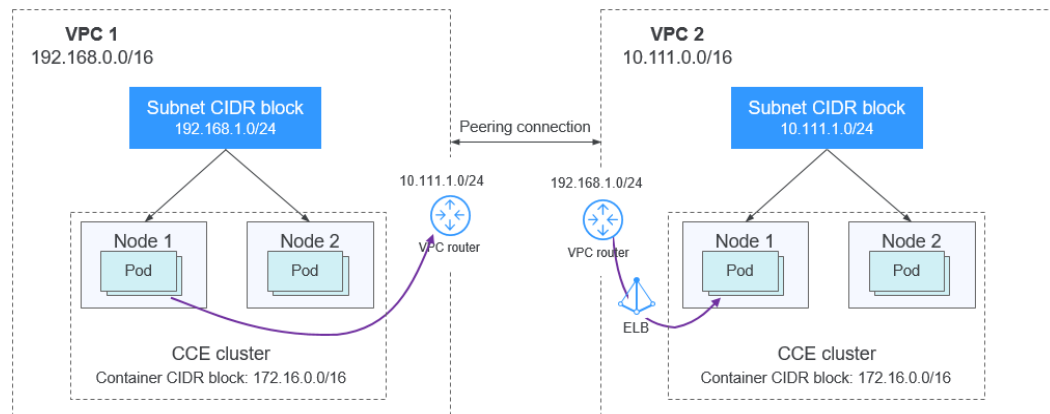
- The VPC to which the clusters belong must not overlap. In each cluster, the subnet CIDR block cannot overlap with the container CIDR block.
- The container CIDR blocks of clusters at both ends cannot overlap, but the Service CIDR blocks can.
- If the request end cluster uses the VPC network, check whether the node security group in the destination cluster allows the container CIDR block of the request end cluster. If yes, pods in one cluster can directly access pods in another cluster through the pod IP address. Similarly, if nodes running in the clusters at the two ends of the VPC peering connection need to access each other, the node security group must allow the VPC CIDR block of the peer cluster.

- You need to add routes for accessing the peer network CIDR block to the VPC routing tables at both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2.
 - **Add the VPC CIDR block of the peer cluster:** After the route of the VPC CIDR block is added, a pod in a cluster can access another cluster node. For example, the pod can access the port of a NodePort Service.
 - **Add peer container CIDR block:** After the route of the container CIDR block is added, a pod can directly access pods in another cluster through the container IP addresses.

Clusters using tunnel networks

To allow clusters that use tunnel networks to access each other across VPCs, add routes to the two ends of the VPC peering after a VPC peering connection is created.

Figure 9-6 Tunnel network - VPC interconnection scenario



Pay attention to the following:

- The VPCs of the peer clusters must not overlap.
- The container CIDR blocks of all clusters can overlap, so do the Service CIDR blocks.
- If the request end cluster uses the tunnel network, check whether the node security group in the destination cluster allows the VPC CIDR block (including the node subnets) of the request end cluster. If yes, nodes in one cluster can access nodes in another cluster. However, pods in different clusters cannot be directly accessed using pod IP addresses. Access between pods in different clusters requires Services. The LoadBlancer Services are recommended.
- The VPC CIDR block route of the peer cluster must be added to the VPC routing tables of both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2. After the route of the VPC CIDR block is added, the pod can access another cluster node, for example, accessing the port of a NodePort Service.

Clusters using different networks

If clusters using different networks need to communicate with each other across VPCs, every one of them may serve as the request end or destination end. Pay attention to the following:

- The VPC CIDR block to which the cluster belongs cannot overlap with the VPC CIDR block of the peer cluster.
- Cluster subnet CIDR blocks cannot overlap with the container CIDR blocks.
- Container CIDR blocks in different clusters cannot overlap with each other.
- If pods or nodes in different clusters need to access each other, the security groups of the clusters on both ends must allow the corresponding CIDR blocks based on the following rules:
 - If the request end cluster uses the VPC network, the node security group of the destination cluster must allow the VPC CIDR block (including the node subnets and container CIDR block) of the request end cluster.
 - If the request end cluster uses the tunnel network, the node security group of the destination cluster must allow the VPC CIDR block (including the node subnets) of the request end cluster.
- The VPC CIDR block route of the peer cluster must be added to the VPC routing tables of both ends. For example, you need to add a route for accessing the CIDR block of VPC 2 to the route table of VPC 1, and add a route for accessing VPC 1 to the route table of VPC 2. After the route of the VPC CIDR block is added, the pod can access another cluster node, for example, accessing the port of a NodePort Service.

If a cluster uses the VPC network, the VPC routing tables at both ends must contain its container CIDR block. After the container CIDR block route is added, the pod can directly access pods in another cluster through the container IP addresses.

VPC-IDC Scenarios

Similar to the VPC interconnection scenario, some CIDR blocks in the VPC are routed to the IDC. The pod IP addresses of CCE clusters cannot overlap with the addresses within these CIDR blocks. To access the pod IP addresses in the cluster in the IDC, configure the route table to the private line VBR on the IDC.

9.2 Selecting a Network Model

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

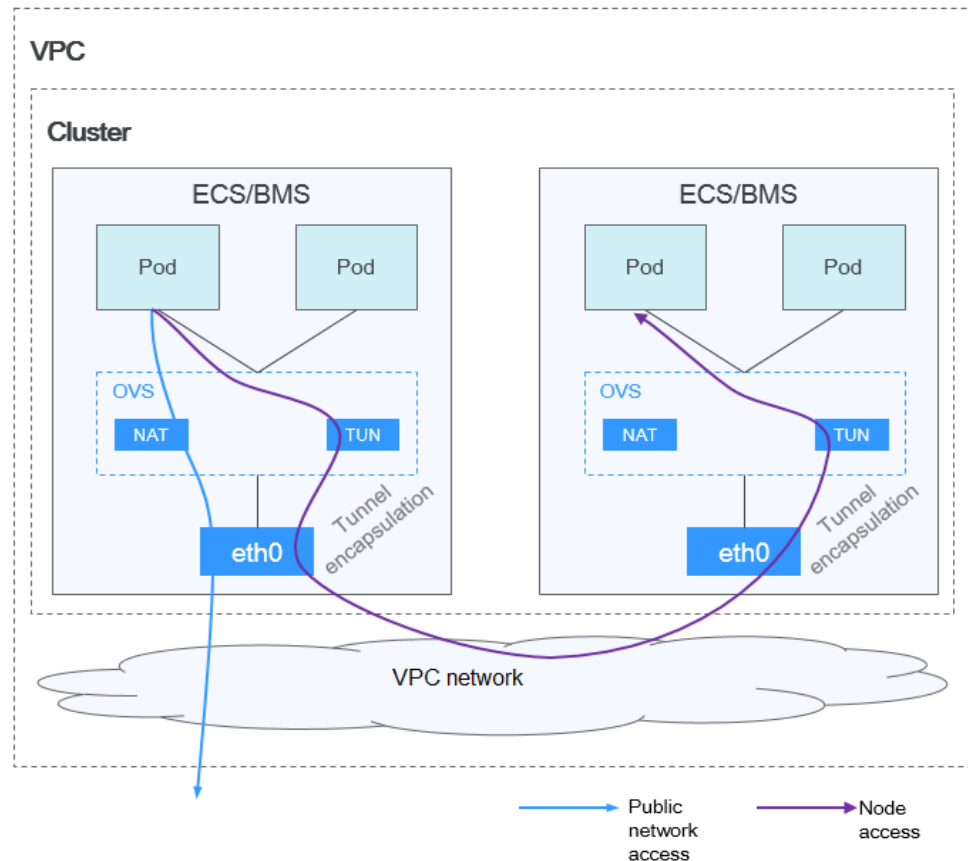
⚠ CAUTION

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.

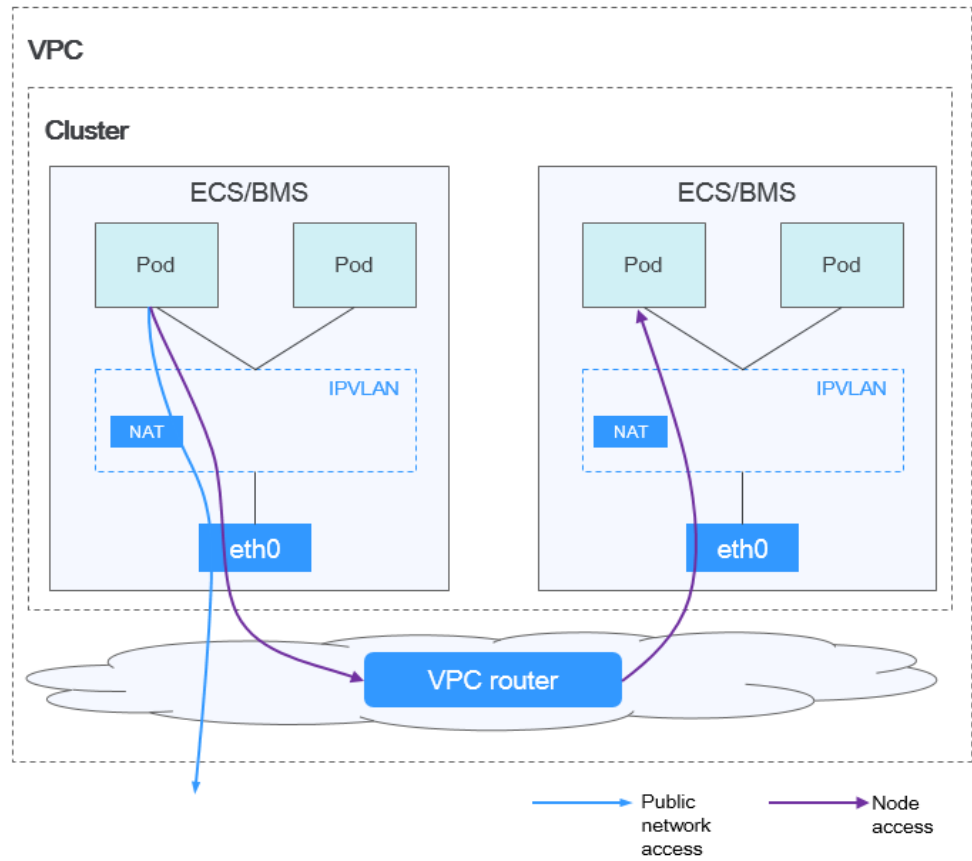
Though at some cost of performance, the tunnel encapsulation enables higher interoperability and compatibility with advanced features (such as network policy-based isolation), meeting the requirements of most applications.

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 Network model comparison

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Application scenarios	<ul style="list-style-type: none"> • Low requirements on performance: As the container tunnel network requires additional VXLAN tunnel encapsulation, it has about 5% to 15% of performance loss when compared with the other two container network models. Therefore, the container tunnel network applies to the scenarios that do not have high performance requirements, such as web applications, and middle-end and back-end services with a small number of access requests. • Large-scale networking: Different from the VPC network that is limited by the VPC route quota, the container tunnel network does not have any restriction 	<ul style="list-style-type: none"> • High performance requirements: As no tunnel encapsulation is required, the VPC network model delivers the performance close to that of a VPC network when compared with the container tunnel network model. Therefore, the VPC network model applies to scenarios that have high requirements on performance, such as AI computing and big data computing. • Small- and medium-scale networks: Due to the limitation on VPC routing tables, it is recommended that the number of nodes in a cluster be less than or equal to 1000. 	<ul style="list-style-type: none"> • High performance requirements: Cloud Native 2.0 networks use VPC networks to construct container networks, eliminating the need for tunnel encapsulation or NAT when containers communicate. This makes Cloud Native 2.0 networks ideal for scenarios that demand high bandwidth and low latency, such as live streaming and e-commerce flash sales. • Large-scale networking: Cloud Native 2.0 networks support a maximum of 2,000 ECS nodes and 100,000 pods.

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
	on the infrastructure. In addition, the container tunnel network controls the broadcast domain to the node level. The container tunnel network supports a maximum of 2000 nodes.		
Core technology	OVS	IPvlan and VPC route	VPC ENI/sub-ENI
Applicable clusters	CCE standard cluster	CCE standard cluster	CCE Turbo cluster
Container network isolation	Kubernetes native NetworkPolicy for pods	No	Pods support security group isolation.
Interconnecting pods to a load balancer	Interconnected through a NodePort	Interconnected through a NodePort	Directly interconnected using a dedicated load balancer Interconnected using a shared load balancer through a NodePort
Managing container IP addresses	<ul style="list-style-type: none"> Separate container CIDR blocks needed Container CIDR blocks divided by node and dynamically added after being allocated 	<ul style="list-style-type: none"> Separate container CIDR blocks needed Container CIDR blocks divided by node and statically allocated (the allocated CIDR blocks cannot be changed after a node is created) 	Container CIDR blocks divided from a VPC subnet (You do not need to configure separate container CIDR blocks.)

Dimension	Tunnel Network	VPC Network	Cloud Native Network 2.0
Network performance	Performance loss due to VXLAN encapsulation	No tunnel encapsulation, and cross-node traffic forwarded through VPC routers (The performance is so good that is comparable to that of the host network, but there is a loss caused by NAT.)	Container network integrated with VPC network, eliminating performance loss
Networking scale	A maximum of 2000 nodes are supported.	Suitable for small- and medium-scale networks due to the limitation on VPC routing tables. It is recommended that the number of nodes be less than or equal to 1000. Each time a node is added to the cluster, a route is added to the VPC routing tables. Evaluate the cluster scale that is limited by the VPC routing tables before creating the cluster.	A maximum of 2000 nodes are supported. In a cloud-native network 2.0 cluster, containers' IP addresses are assigned from VPC CIDR blocks, and the number of containers supported is restricted by these blocks. Evaluate the cluster's scale limitations before creating it.

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

Sticky sessions ensure continuity and consistency when you access applications. If a load balancer is deployed between a client and backend servers, connections may be forwarded to different servers for processing. Sticky sessions can resolve this issue. After sticky session is enabled, requests from the same client will be continuously distributed to the same backend server through load balancing.

For example, in most online systems that require user identity authentication, a user needs to interact with the server for multiple times to complete a session. These interactions require continuity. If sticky session is not configured, the load balancer may allocate certain requests to different backend servers. Since user identity has not been authenticated on other backend servers, interaction exceptions such as a user login failure may occur.

Therefore, select a proper sticky session type based on the application environment.

Table 9-2 Sticky session types

OSI Layer	Listener Protocol and Networking	Sticky Session Type	Scenarios Where Sticky Sessions Become Invalid
Layer 4	TCP- or UDP-compliant Services	Source IP address: The source IP address of each request is calculated using the consistent hashing algorithm to obtain a unique hashing key, and all backend servers are numbered. The system allocates the client to a particular server based on the generated key. This allows requests from the same IP address are forwarded to the same backend server.	<ul style="list-style-type: none"> • Source IP addresses of the clients have changed. • Requests from the clients exceed the session stickiness duration.

OSI Layer	Listener Protocol and Networking	Sticky Session Type	Scenarios Where Sticky Sessions Become Invalid
Layer 7	HTTP- or HTTPS-compliant ingresses	<ul style="list-style-type: none"> • Load balancer cookie: The load balancer generates a cookie after receiving a request from the client. All subsequent requests with the cookie will be routed to the same backend server. • Application cookie: The application deployed on the backend server generates a cookie after receiving the first request from the client. All subsequent requests with the same cookie will be routed to the same backend server. 	<ul style="list-style-type: none"> • If requests sent by the clients do not contain a cookie, sticky sessions will not take effect. • Requests from the clients exceed the session stickiness duration.

 **NOTE**

When creating a load balancer, configure sticky sessions by setting `kubernetes.io/elb.lb-algorithm` to `ROUND_ROBIN` or `kubernetes.io/elb.lb-algorithm` to `LEAST_CONNECTIONS`. If you set `kubernetes.io/elb.lb-algorithm` is to `SOURCE_IP`, source IP address-based sticky sessions are supported. In this case, you do not need to configure sticky sessions again.

Layer 4 Sticky Sessions for Services

In Layer 4 mode, source IP address-based sticky sessions can be enabled, where hash routing is performed based on the client IP address.

Enabling Layer 4 Sticky Session in a CCE Standard Cluster

In a CCE standard cluster, to enable source IP address-based sticky session for a Service, ensure the following conditions are met:

1. **Service Affinity** of the Service must be set to **Node-level**, where the `externalTrafficPolicy` value of the Service must be **Local**.
2. Anti-affinity has been enabled on the backend applications of the Service to prevent all pods from being deployed on the same node.

Procedure

Step 1 Create an Nginx workload.

Set the number of pods to 3 and configure podAntiAffinity.

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

```
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
          imagePullSecrets:
            - name: default-secret
      affinity:
        podAntiAffinity:
          # Pod anti-affinity
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              topologyKey: kubernetes.io/hostname
```

Step 2 Create a LoadBalancer Service, for example, using an existing load balancer. The following shows an example YAML file for configuring source IP address-based sticky sessions:

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: union
    kubernetes.io/elb.id: *****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP # Enable source IP address-based sticky session.
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Local # Node level Service affinity
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

Step 3 Check whether the Layer 4 sticky session function is enabled.

1. Log in to the ELB console, locate the row containing the target load balancer, and click the listener name.
2. Check whether the sticky session function is enabled in the backend server group.

----End

Enabling Layer 4 Sticky Session in a CCE Turbo Cluster

In a CCE Turbo cluster, enabling source IP address-based sticky session for a Service relies on the load balancer type.

- When a dedicated load balancer is used, passthrough networking is allowed between the load balancer and pods, and pods function as the backend server group of the load balancer. Therefore, you do not need to configure Service affinity or application anti-affinity when enabling source IP address-based sticky session for the Service.
- If a shared load balancer is used, sticky session cannot be enabled.

Procedure

- **For dedicated load balancers**

The following shows an example YAML file for configuring source IP address-based sticky sessions for a Service that uses an existing load balancer:

```
apiVersion: v1
kind: Service
metadata:
  name: svc-example
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.id: ****
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: SOURCE_IP # Enable source IP address-based sticky
session.
spec:
  selector:
    app: nginx
  externalTrafficPolicy: Cluster # In CCE Turbo clusters, Service affinity does not need to be
configured if a dedicated load balancer is used.
  ports:
    - name: cce-service-0
      targetPort: 80
      nodePort: 32633
      port: 80
      protocol: TCP
  type: LoadBalancer
```

Verify that the Layer 4 sticky session function is enabled.

- a. Log in to the ELB console, locate the row containing the target load balancer, and click the listener name.
- b. Check whether the sticky session function is enabled in the backend server group.

Layer 7 Sticky Sessions for Ingresses

In Layer 7 mode, sticky sessions can be enabled using HTTP cookies or application cookies.

Enabling Layer 7 Sticky Session in a CCE Standard Cluster

To enable cookie-based sticky session on an ingress, ensure the following conditions are met:

1. **Service Affinity** of the ingress must be set to **Node-level**, where the **externalTrafficPolicy** value of the Service must be **Local**.
2. Anti-affinity must be enabled for the ingress workload to prevent all pods from being deployed on the same node.

Procedure

Step 1 Create an Nginx workload.

Set the number of pods to 3 and configure podAntiAffinity.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: 'nginx:perl'
          resources:
            limits:
              cpu: 250m
              memory: 512Mi
            requests:
              cpu: 250m
              memory: 512Mi
          imagePullSecrets:
            - name: default-secret
      affinity:
        podAntiAffinity:
          # Pod anti-affinity
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - nginx
              topologyKey: kubernetes.io/hostname
```

Step 2 Create a Service for the workload. This section uses a NodePort Service as an example.

Configure sticky sessions during the creation of a Service. An ingress can access multiple Services, and each Service can have different sticky sessions.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
annotations:
  kubernetes.io/elb.lb-algorithm: ROUND_ROBIN # Weighted round robin allocation policy
  kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE # HTTP cookie
  kubernetes.io/elb.session-affinity-option: '{"persistence_timeout":"1440"}' # Session stickiness duration,
in minutes. The value ranges from 1 to 1440.
spec:
  selector:
    app: nginx
  ports:
```

```
- name: cce-service-0
  protocol: TCP
  port: 80
  targetPort: 80
  nodePort: 32633      # Custom node port
  type: NodePort
  externalTrafficPolicy: Local # Node level Service affinity
```

You can also select **APP_COOKIE**.

NOTICE

Only shared load balancers support application cookie-based sticky sessions.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.algorithm: ROUND_ROBIN # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: APP_COOKIE # Select APP_COOKIE.
    kubernetes.io/elb.session-affinity-option: '{"app_cookie_name":"test"}' # Application cookie name
spec:
  selector:
    app: nginx
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 32633      # Custom node port
      type: NodePort
      externalTrafficPolicy: Local # Node level Service affinity
```

Step 3 Create an ingress and associate it with the 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.id: *****
spec:
  rules:
    - host: 'www.example.com'
      http:
        paths:
          - path: '/'
            backend:
              service:
                name: nginx # Service name
                port:
                  number: 80
              property:
                ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
          ingressClassName: cce
```

Step 4 Verify that the Layer 7 sticky session function is enabled.

1. Log in to the ELB console, locate the row containing the target load balancer, and click the listener name.

2. Click the **Forwarding Policies** tab, click the backend server group name, and check whether sticky session is enabled for it.

----End

Enabling Layer 7 Sticky Session in a CCE Turbo Cluster

Enable cookie-based sticky session on the ingress.

- When a dedicated load balancer is used, passthrough networking is allowed between the load balancer and pods, and pods function as the backend server group of the load balancer. Therefore, you do not need to configure Service affinity or application anti-affinity when enabling cookie-based sticky session for the ingress.
- If a shared load balancer is used, sticky session cannot be enabled.

Procedure

- **For dedicated load balancers**

- a. Create a Service for the workload. In a CCE Turbo cluster, the ingresses that use a dedicated load balancer must interconnect with ClusterIP Services.

Configure sticky sessions during the creation of a Service. An ingress can access multiple Services, and each Service can have different sticky sessions.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
  annotations:
    kubernetes.io/elb.lb-algorithm: ROUND_ROBIN # Weighted round robin allocation policy
    kubernetes.io/elb.session-affinity-mode: HTTP_COOKIE # HTTP cookie
    kubernetes.io/elb.session-affinity-option: '{"persistence_timeout":"1440"}' # Session
stickiness duration, in minutes. The value ranges from 1 to 1440.
spec:
  selector:
    app: nginx
  ports:
    - name: cce-service-0
      protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 0
  type: ClusterIP
```

- b. Create an ingress and associate it with the Service.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test
  namespace: default
  annotations:
    kubernetes.io/elb.class: performance
    kubernetes.io/elb.port: '80'
    kubernetes.io/elb.id: *****
spec:
  rules:
    - host: 'www.example.com'
      http:
        paths:
          - path: '/'
```

```
backend:
  service:
    name: nginx    # Service name
    port:
      number: 80
  property:
    ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
    pathType: ImplementationSpecific
  ingressClassName: cce
```

- c. Verify that the Layer 7 sticky session function is enabled.
 - i. Log in to the ELB console, locate the row containing the target load balancer, and click the listener name.
 - ii. Click the **Forwarding Policies** tab, click the backend server group name, and check whether sticky session is enabled for it.

9.4 Pre-Binding Container ENI for CCE Turbo Clusters

In the Cloud Native 2.0 network model, each pod is allocated an ENI or a sub-ENI (called container ENI). The speed of ENI creation and binding is slower than that of pod scaling, severely affecting the container startup speed in large-scale batch creation. Therefore, the Cloud Native Network 2.0 model provides the dynamic pre-binding of container ENIs to accelerate pod startup while improving IP resource utilization.

Constraints

- CCE Turbo clusters of 1.23.5-r0, 1.25.1-r0, or later support ENI pre-binding, global configuration at the cluster level, and custom settings at the node pool level. Custom settings of nodes out of a node pool is not supported.
- 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 9-3 Parameters of the dynamic ENI pre-binding policy

Parameter	Default Value	Description	Suggestion
nic-minimum-target	10	<p>Minimum number of container ENIs bound to a node.</p> <p>The parameter value must be a positive integer. The value 10 indicates that there are at least 10 container ENIs bound to a node. If the number you entered exceeds the container ENI quota of the node, the ENI quota will be used.</p>	Configure these parameters based on the number of pods.
nic-maximum-target	0	<p>If the number of ENIs bound to a node exceeds the value of nic-maximum-target, the system does not proactively pre-bind ENIs.</p> <p>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.</p> <p>The parameter value must be a positive integer. The value 0 indicates that the check on the upper limit of pre-bound container ENIs is disabled. If the number you entered exceeds the container ENI quota of the node, the ENI quota will be used.</p>	Configure these parameters based on the number of pods.
nic-warm-target	2	<p>Minimum number of pre-bound ENIs on a node. The value must be a number.</p> <p>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.</p>	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	<p>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.</p> <ul style="list-style-type: none"> 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. 	<p>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.</p>

Configuration Example

Level	Service Scenario	Configuration Example
Cluster	<p>All nodes use the c7.4xlarge.2 model (sub-ENI quota: 128).</p> <p>Most nodes run about 20 pods.</p> <p>Most nodes can run a maximum of 60 pods.</p> <p>Most nodes can scale out 10 pods within 10 seconds.</p> <p>Most nodes frequently scale in or out 15 pods within minutes.</p>	<p>Cluster-level global configuration:</p> <ul style="list-style-type: none"> nic-minimum-target: 20 nic-maximum-target: 60 nic-warm-target: 10 nic-max-above-warm-target: 5

Level	Service Scenario	Configuration Example
Node pool	<p>A node pool that uses the c7.xlarge.2 high-specification model is created in the cluster. (sub-ENI quota: 256)</p> <p>Most nodes run about 100 pods.</p> <p>Most nodes can run a maximum of 128 pods.</p> <p>Most nodes can scale out 10 pods within 10 seconds.</p> <p>Most nodes frequently scale in or out 12 pods within minutes.</p>	<p>Custom settings at the node pool level:</p> <ul style="list-style-type: none"> • nic-minimum-target: 100 • nic-maximum-target: 120 • nic-warm-target: 10 • nic-max-above-warm-target: 2

 **NOTE**

Pods using HostNetwork are excluded.

Cluster-level Global Configuration

Step 1 Log in to the CCE console. In the navigation pane, choose **Clusters**.

Step 2 Click  next to the target cluster and choose **Manage**.

Step 3 In the window that slides out from the right, click **Networking Components**. For details about the parameter configurations, see [Configuration Example](#).

Step 4 After the configuration is complete, click **OK**. Wait for about 10 seconds for the configuration to take effect.

----End

Custom Settings at the Node Pool Level

Step 1 Log in to the CCE console.

Step 2 Click the cluster name to access the cluster console, choose **Nodes** in the navigation pane, and click the **Node Pools** tab.

Step 3 Locate the row containing the target node pool and click **Manage**.

Step 4 In the window that slides out from the right, click **Networking Components** and enable node pool container ENI pre-binding. For details about the parameter configurations, see [Configuration Example](#).

Step 5 After the configuration is complete, click **OK**. Wait for about 10 seconds for the configuration to take effect.

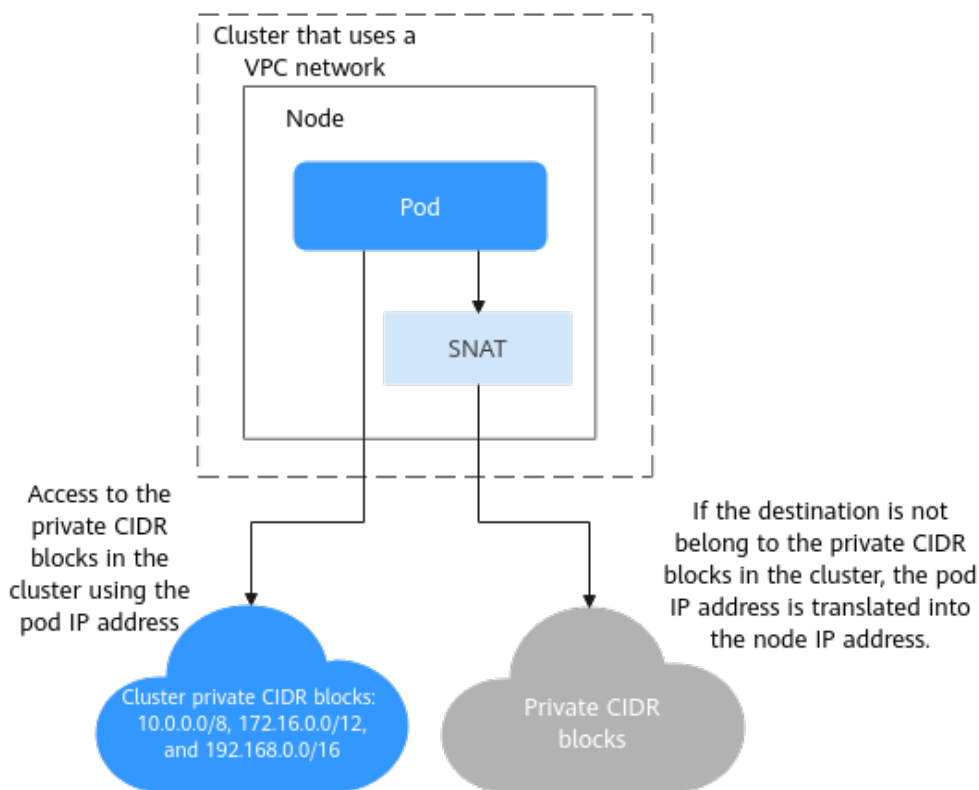
----End

9.5 Accessing an IP Address Outside a Cluster That Uses a VPC Network Using Source Pod IP Addresses in the Cluster

In a CCE cluster that uses a VPC network, when pods try to communicate with external systems, CCE automatically translates the source IP addresses of the pods into the IP addresses of the nodes that are running them. This allows pods to communicate with external systems using the node IP addresses. This process is known as pod IP address masquerading or Source Network Address Translation (SNAT).

You are allowed to configure private CIDR blocks for your clusters using the **nonMasqueradeCIDRs** parameter. If a pod tries to access a private CIDR block, the source node will not perform NAT on the pod IP address. Instead, the VPC route table can directly send the pod data packet to the destination, which means, the pod IP address is directly used to communicate with the private CIDR block in the cluster.

Figure 9-9 Pod IP address translation



Prerequisites

You have a cluster that uses the VPC network and whose version is v1.23.14-r0, v1.25.9-r0, v1.27.6-r0, v1.28.4-r0, or later.

Default Non-Masqueraded CIDR Block Settings in a CCE Cluster

By default, CCE uses the following well-known private CIDR blocks as non-masqueraded CIDR blocks in each cluster:

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

Additionally, in a CCE cluster that uses a secondary VPC CIDR block, adding or resetting a node will automatically include the secondary CIDR block in the non-masqueraded CIDR blocks.

This means that when a pod communicates with external resources and accesses these CIDR blocks, the source IP address of the data packet remains unchanged and is not translated into the node IP address.

Scenarios Where the Default Non-Masqueraded CIDR Blocks Do Not Fit

The default non-masqueraded CIDR block settings in CCE clusters apply to typical scenarios, but in certain specific scenarios, these default settings may not be sufficient to meet user requirements. The following shows typical examples:

- Cross-node access to pods in a cluster

When a node in a Kubernetes cluster needs to access a pod on another node, the response data packet sent from the pod is automatically subject to SNAT. This changes the source IP address from the pod IP address to the IP address of the node that runs the pod. However, this automatic IP address translation can sometimes lead to communication issues, making cross-node access impossible.

To enable a node to access pods on other nodes, you can add the CIDR block of the subnet where the node is located to the **nonMasqueradeCIDRs** parameter. This will skip SNAT and allow the original IP addresses of pods on these nodes to be retained.

- Access from other resources in the same VPC as a cluster to pods in the cluster

In certain scenarios, it may be necessary to access the original IP addresses of pods on different nodes in a CCE cluster directly from other resources (such as ECSs) in the same VPC as the cluster. However, with SNAT enabled by default, the source IP addresses of the data packets are replaced with the IP addresses of the nodes that run these pods when the data packets pass through the nodes. This makes it difficult for these resources to access pods directly.

To enable direct access from resources in the same VPC as the cluster to pods, you can add the CIDR blocks of the subnets where these resources are located to the **nonMasqueradeCIDRs** parameter. This will skip SNAT and ensure that the source IP addresses of the data packets remain the same as the original IP addresses of pods.

Precautions

If a security group or ACL is configured for a cloud service and only the IP address of the node where the pod runs is allowed to access the service, SNAT is required to translate the pod IP address into the node IP address for successful access. As a

result, the CIDR block of the subnet where the server is located cannot be added to the **nonMasqueradeCIDRs** configuration.

The default setting of pod IP address masquerading (SNAT) is usually sufficient. However, if you need to retain the original IP addresses of pods in specific scenarios, you can configure the **nonMasqueradeCIDRs** parameter.

Before doing so, make sure you have evaluated your application scenario and understood the potential risks of improper configuration, because it may block access within clusters. If you are unsure whether to configure this parameter, it is recommended that you keep the default settings and adjust the configuration later once the requirement is clarified.

Procedure

To reserve the source IP address of a pod when the pod accesses a CIDR block, you can configure **nonMasqueradeCIDRs** to specify the CIDR block that does not need to be masqueraded.

- Step 1** Log in to the CCE console and click the name of the target cluster to access the cluster console.
 - Step 2** In the navigation pane, choose **Settings** and click the **Network** tab.
 - Step 3** Modify the range of the CIDR block for non-masquerading access to preserve the source pod IP address when accessing a specified CIDR block. Make sure the parameter configuration complies with the following rules:
 - Each CIDR block must comply with the CIDR format and must be a valid IPv4 CIDR block.
Example of a correct CIDR block: **192.168.1.0/24**
Example of an incorrect CIDR block: **192.168.1.1/24** (incompliant with the CIDR format)
 - The CIDR blocks you configured do not overlap with each other.
Example of correct CIDR blocks: **192.168.1.0/24** and **192.168.2.0/24**
Example of incorrect CIDR blocks: **192.168.1.0/24** and **192.168.1.128/25** (The two CIDR blocks overlap.)
 - The **nonMasqueradeCIDRs** parameter must contain all destination CIDR blocks that you want them to use the original pod IP addresses for communications.
 - Step 4** After the modification, click **Confirm configuration**. The setting takes effect within 1 minute.
- 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	The first data disk attached to a node for container engine and kubelet	<ul style="list-style-type: none"> • Expanding the Container Engine Capacity • Expanding the kubelet Capacity • Expanding Capacity of the Disk Shared by Container Engine and kubelet
Container storage	Pod container space	The base size of a container, which is, the upper limit of the disk space occupied by each pod (including the storage space occupied by container images)	Expanding the Capacity of a Data Disk Used by Pod (basesize)
	PVC	Storage resources mounted to the containers	Expanding a PVC

Expanding System Disk Capacity

EulerOS 2.9 is used as the sample OS. There is only one partition (**/dev/vda1**) with a capacity of 50 GiB in the system disk **/dev/vda**, and then 50 GiB is added to the system disk. In this example, the additional 50 GiB is allocated to the existing **/dev/vda1** partition.

Step 1 Expand the system disk capacity on the EVS console.

Only the storage capacity of the EVS disk is expanded. You also need to perform the following steps to expand the partition and file system.

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 Data Disk Capacity

The first data disk of a CCE node is composed of container engine and kubelet space by default. If either of them reaches full capacity, you can expand the disk space as needed.

In clusters of v1.21.10-r0, v1.23.8-r0, v1.25.3-r0, and later, CCE enables container engine (Docker/containerd) and kubelet to share the space of the first data disk. If the shared disk space is insufficient, you can expand it.

Expanding the Container Engine Capacity

The available container engine space affects image pulls and container startup and running. This section uses containerd as an example to describe how to expand the container engine capacity.

Step 1 Expand the capacity of a data disk on the EVS console.

Only the storage capacity of the EVS disk is expanded. You also need to perform the following steps to expand the capacity of the logical volume and file system.

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 **dockersys**.

1. Check the disk and partition sizes of the device.

```
# 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 150G  0 disk # The data disk has been expanded to 150 GiB, but 50 GiB
space is not allocated.
└─vgpaas-dockersys 253:0  0  90G  0 lvm  /var/lib/containerd
└─vgpaas-kubernetes 253:1  0  10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

2. Expand the disk capacity.

Add the new disk capacity to the **dockersys** logical volume used by the container engine.

a. Expand the PV capacity so that LVM can identify the new EVS capacity. **/dev/sdb** specifies the physical volume where dockersys is located.

```
pvresize /dev/sdb
```

Information similar to the following is displayed:

```
Physical volume "/dev/sdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

b. Expand 100% of the free capacity to the logical volume. **vgpaas/dockersys** specifies the logical volume used by the container engine.

```
lvextend -l+100%FREE -n vgpaas/dockersys
```

Information similar to the following is displayed:

```
Size of logical volume vgpaas/dockersys changed from <90.00 GiB (23039 extents) to 140.00
GiB (35840 extents).
Logical volume vgpaas/dockersys successfully resized.
```

c. Adjust the size of the file system. **/dev/vgpaas/dockersys** specifies the file system path of the container engine.

```
resize2fs /dev/vgpaas/dockersys
```

Information similar to the following is displayed:

```
Filesystem at /dev/vgpaas/dockersys is mounted on /var/lib/containerd; on-line resizing required
old_desc_blocks = 12, new_desc_blocks = 18
The filesystem on /dev/vgpaas/dockersys is now 36700160 blocks long.
```

3. Check whether the capacity is expanded.

```
# 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 150G  0 disk
├─vgpaas-dockersys 253:0 0 140G  0 lvm  /var/lib/containerd
└─vgpaas-kubernetes 253:1 0 10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

Devicemapper: A thin pool is allocated to store image data.

1. Check the disk and partition sizes of the device.

```
# lsblk
NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda           8:0  0  50G  0 disk
├─vda1        8:1  0  50G  0 part /
vdb           8:16 0 200G  0 disk
├─vgpaas-dockersys 253:0 0 18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta 253:1 0 3G  0 lvm
├─vgpaas-thinpool 253:3 0 67G  0 lvm          # Space used by thinpool
├─...
├─vgpaas-thinpool_tdata 253:2 0 67G  0 lvm
├─vgpaas-thinpool 253:3 0 67G  0 lvm
├─...
└─vgpaas-kubernetes 253:4 0 10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

2. Expand the disk capacity.

Option 1: Add the new disk capacity to the thin pool disk.

- a. Expand the PV capacity so that LVM can identify the new EVS capacity. `/dev/vdb` specifies the physical volume where thinpool is located. `pvresize /dev/vdb`

Information similar to the following is displayed:

```
Physical volume "/dev/vdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

- b. Expand 100% of the free capacity to the logical volume. `vgpaas/thinpool` specifies the logical volume used by the container engine. `lvextend -l+100%FREE -n vgpaas/thinpool`

Information similar to the following is displayed:

```
Size of logical volume vgpaas/thinpool changed from <67.00 GiB (23039 extents) to <167.00 GiB (48639 extents).
Logical volume vgpaas/thinpool successfully resized.
```

- c. Do not need to adjust the size of the file system, because the thin pool is not mounted to any devices.
- d. Check whether the capacity is expanded. Run the `lsblk` command to check the disk and partition sizes of the device. If the new disk capacity has been added to the thin pool, the capacity is expanded.

```
# lsblk
NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda           8:0  0  50G  0 disk
├─vda1        8:1  0  50G  0 part /
vdb           8:16 0 200G  0 disk
├─vgpaas-dockersys 253:0 0 18G  0 lvm  /var/lib/docker
├─vgpaas-thinpool_tmeta 253:1 0 3G  0 lvm
├─vgpaas-thinpool 253:3 0 167G  0 lvm          # Thin pool space after
capacity expansion
├─...
├─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
```

Option 2: Add the new disk capacity to the `dockersys` disk.

- a. Expand the PV capacity so that LVM can identify the new EVS capacity. `/dev/vdb` specifies the physical volume where dockersys is located.

```
pvresize /dev/vdb
```

Information similar to the following is displayed:

```
Physical volume "/dev/vdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

- b. Expand 100% of the free capacity to the logical volume. `vgpaas/dockersys` specifies the logical volume used by the container engine.

```
lvextend -l+100%FREE -n vgpaas/dockersys
```

Information similar to the following is displayed:

```
Size of logical volume vgpaas/dockersys changed from <18.00 GiB (4607 extents) to <118.00 GiB (30208 extents).
Logical volume vgpaas/dockersys successfully resized.
```

- c. Adjust the size of the file system. `/dev/vgpaas/dockersys` specifies the file system path of the container engine.

```
resize2fs /dev/vgpaas/dockersys
```

Information similar to the following is displayed:

```
Filesystem at /dev/vgpaas/dockersys is mounted on /var/lib/docker; on-line resizing required
old_desc_blocks = 3, new_desc_blocks = 15
The filesystem on /dev/vgpaas/dockersys is now 30932992 blocks long.
```

- d. Check whether the capacity is expanded. Run the `lsblk` command to check the disk and partition sizes of the device. If the new disk capacity has been added to the dockersys, the capacity is expanded.

```
# lsblk
NAME                                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda                                  8:0  0  50G  0 disk
├─vda1                                8:1  0  50G  0 part /
└─vdb                                  8:16 0 200G  0 disk
   └─vgpaas-dockersys                 253:0  0 118G  0 lvm  /var/lib/docker # dockersys after
      capacity expansion
         └─vgpaas-thinpool_tmeta       253:1  0   3G  0 lvm
            └─vgpaas-thinpool         253:3  0  67G  0 lvm
               ...
         └─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
```

----End

Expanding the kubelet Capacity

The kubelet space serves as a temporary storage location for kubelet components and EmptyDir. You can follow the following steps to increase the kubelet capacity:

- Step 1** Expand the capacity of a data disk on the EVS console.

Only the storage capacity of the EVS disk is expanded. You also need to perform the following steps to expand the capacity of the logical volume and file system.

- Step 2** Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More > Sync Server Data** in the row containing the target node.

- Step 3** Log in to the target node.

- Step 4** Run `lsblk` to view the block device information of the node.

```
# lsblk
NAME                                MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
```

```
sda            8:0  0  50G  0 disk
└─sda1         8:1  0  50G  0 part /
sdb          8:16  0  200G  0 disk  #The data disk has been expanded to 200 GiB, but 50 GiB space
is not allocated.
└─vgpaas-dockersys 253:0  0  140G  0 lvm  /var/lib/containerd
   └─vgpaas-kubernetes 253:1  0  10G  0 lvm  /mnt/paas/kubernetes/kubelet
```

Step 5 Perform the following operations on the node to add the new disk capacity to the kubelet space:

1. Expand the PV capacity so that LVM can identify the new EVS capacity. `/dev/sdb` specifies the physical volume where kubelet is located.
`pvresize /dev/sdb`

Information similar to the following is displayed:

```
Physical volume "/dev/sdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

2. Expand 100% of the free capacity to the logical volume. `vgpaas/kubernetes` specifies the logical volume used by kubelet.
`lvextend -l+100%FREE -n vgpaas/kubernetes`

Information similar to the following is displayed:

```
Size of logical volume vgpaas/kubernetes changed from <10.00 GiB (2559 extents) to <60.00 GiB
(15359 extents).
Logical volume vgpaas/kubernetes successfully resized.
```

3. Adjust the size of the file system. `/dev/vgpaas/kubernetes` specifies the file system path of the container engine.
`resize2fs /dev/vgpaas/kubernetes`

Information similar to the following is displayed:

```
Filesystem at /dev/vgpaas/kubernetes is mounted on /mnt/paas/kubernetes/kubelet; on-line resizing
required
old_desc_blocks = 2, new_desc_blocks = 8
The filesystem on /dev/vgpaas/kubernetes is now 15727616 blocks long.
```

Step 6 Run `lsblk` to view the block device information of the node.

```
# 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  140G  0 lvm  /var/lib/containerd
   └─vgpaas-kubernetes 253:1  0  60G  0 lvm  /mnt/paas/kubernetes/kubelet # Allocate the new disk to
the kubelet space.
```

----End

Expanding Capacity of the Disk Shared by Container Engine and kubelet

To expand the capacity of the disk shared by container engine and kubelet, perform the following steps:

Step 1 Expand the capacity of a data disk on the EVS console.

Only the storage capacity of the EVS disk is expanded. You also need to perform the following steps to expand the capacity of the logical volume and file system.

Step 2 Log in to the CCE console and click the cluster. In the navigation pane, choose **Nodes**. Click **More > Sync Server Data** in the row containing the target node.

Step 3 Log in to the target node.

Step 4 Run `lsblk` to view the block device information of the node.

```
# lsblk
NAME        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda          8:0    0  50G  0 disk
└─sda1       8:1    0  50G  0 part /
sdb          8:16   0 120G  0 disk # The data disk has been expanded to 120 GiB, but 20 GiB space is not
allocated.
└─vgpaas-share 253:0   0 100G  0 lvm  /mnt/paas # Space used by the container engine and the kubelet
component
```

Step 5 Run the following commands on the node to add the new disk capacity to the shared disk:

1. Expand the PV capacity so that LVM can identify the new EVS capacity. `/dev/sdb` specifies the physical volume where the shared disk is located.

```
pvresize /dev/sdb
```

Information similar to the following is displayed:

```
Physical volume "/dev/sdb" changed
1 physical volume(s) resized or updated / 0 physical volume(s) not resized
```

2. Expand 100% of the free capacity to the logical volume. `vgpaas/share` specifies the logical volume shared by the container engine and the kubelet component.

```
lvextend -l+100%FREE -n vgpaas/share
```

Information similar to the following is displayed:

```
Size of logical volume vgpaas/share changed from <100.00 GiB (25599 extents) to <120.00 GiB
(30719 extents).
Logical volume vgpaas/share successfully resized.
```

3. Adjust the size of the file system. `/dev/vgpaas/share` specifies the file system path of the shared disk.

```
resize2fs /dev/vgpaas/share
```

Information similar to the following is displayed:

```
Filesystem at /dev/vgpaas/share is mounted on /mnt/paas; on-line resizing required
old_desc_blocks = 13, new_desc_blocks = 15
The filesystem on /dev/vgpaas/share is now 31456256 blocks long.
```

Step 6 Run `lsblk` to view the block device information of the node.

```
# 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 120G  0 disk
└─vgpaas-share 253:0   0 120G  0 lvm  /mnt/paas # Space of the new disk used by the container engine
and the kubelet component
```

----End

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

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

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

Step 3 Click the **Nodes** tab, locate the row containing the target node, and choose **More** > **Reset Node** in the **Operation** column.

NOTICE

Resetting a node may make the node-specific resources (such as local storage and workloads scheduled to this node) unavailable. Exercise caution when performing this operation to avoid impact on running services.

Step 4 Reconfigure node parameters.

If you need to adjust the container storage space, pay attention to the following configurations:

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

Space Allocation for Pods: indicates the base size of a pod. It is the maximum size that a workload's pods (including the container images) can grow to in the disk space. Proper settings can prevent pods from taking all the disk space available and avoid service exceptions. It is recommended that the value is less than or equal to 80% of the container engine space. This parameter is related to the node OS and container storage rootfs and is not supported in some scenarios.

Step 5 After the node is reset, log in to the node and check whether the container capacity has been expanded. The command output varies with the container storage rootfs.

- **Overlayfs:** No independent thin pool is allocated. Image data is stored in **dockersys**. Run the following command to check whether the container capacity has been expanded:

docker exec -it container_id /bin/sh or **kubectrl exec -it container_id /bin/sh**
df -h

If the information similar to the following is displayed, the overlay capacity has been expanded from 10 GiB to 15 GiB.

```
Filesystem      Size  Used Avail Use% Mounted on
overlay         15G  104K   15G   1% /
tmpfs           64M   0   64M   0% /dev
tmpfs           3.6G   0   3.6G   0% /sys/fs/cgroup
/dev/mapper/vgpaas-share 98G  4.0G   89G   5% /etc/hosts
...
```

- **Devicemapper:** A thin pool is allocated to store image data. Run the following command to check whether the container capacity has been expanded:

docker exec -it container_id /bin/sh or **kubectrl exec -it container_id /bin/sh**
df -h

If the information similar to the following is displayed, the thin pool capacity has been expanded from 10 GiB to 15 GiB.

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vgpaas-thinpool-snap-84 15G  232M   15G   2% /
tmpfs           64M   0   64M   0% /dev
tmpfs           3.6G   0   3.6G   0% /sys/fs/cgroup
/dev/mapper/vgpaas-kubernetes 11G  41M   11G   1% /etc/hosts
/dev/mapper/vgpaas-dockersys 20G  1.1G   18G   6% /etc/hostname
...
```

----End

Expanding a PVC

Cloud storage:

- OBS and SFS: There is no storage restriction and capacity expansion is not required.
- EVS:
 - You can expand the capacity of automatically created pay-per-use volumes on the console. The procedure is as follows:
 - i. Choose **Storage** in the navigation pane. In the right pane, click the **PVCs** tab. Click **More** in the **Operation** column of the target PVC and select **Scale-out**.
 - ii. Enter the capacity to be added and click **OK**.
- SFS Turbo: You can expand the capacity on the SFS console and then change the capacity in the PVC.

10.2 Mounting Object Storage Across Accounts

Application Scenarios

- Cross-account data sharing. For example, multiple teams within a company need to share data, but each team uses a different account.
- Cross-account data migration and backup. When account A is about to be disabled, all data stored in the account needs to be transferred to a new account (account B).
- Data processing and analysis. For example, account B is an external data processor and needs to access raw data from account A to perform tasks such as big data analysis and machine learning.

By linking object storage across accounts, you can share data, lower storage and transmission expenses, and guarantee data security and consistency. This enables various teams or organizations to securely and conveniently access each other's data resources, eliminating the need for repeated storage and redundant transmission. Additionally, data is kept current and compliant, enhancing overall service efficiency and security.

Procedure

Assume that account B needs to access and use an OBS bucket of account A. For details, see [Figure 10-1](#) and [Table 10-2](#).

Figure 10-1 Mounting an OBS bucket across accounts

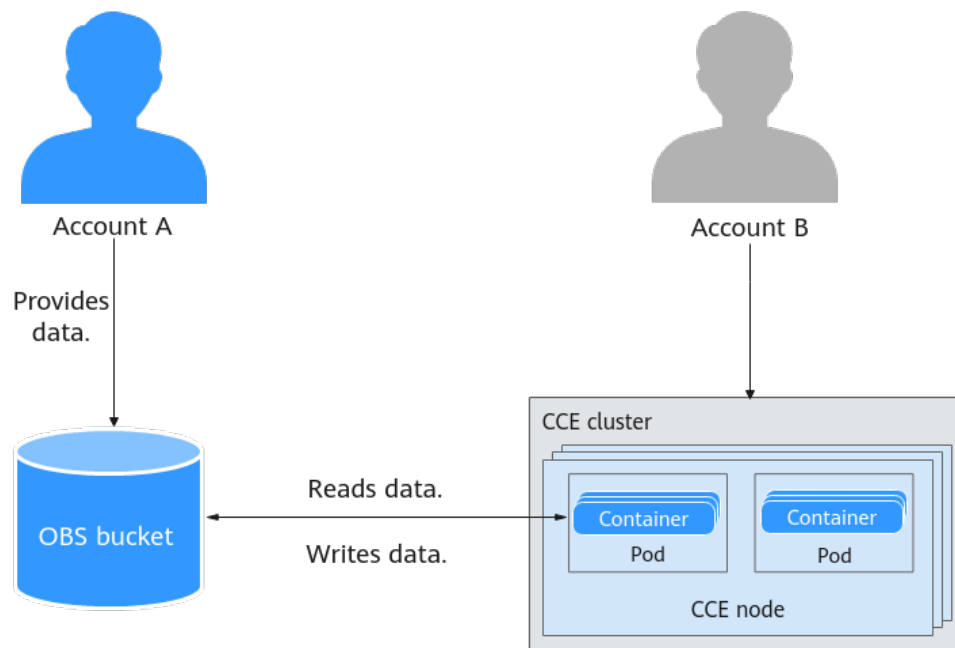


Table 10-2 Process description

Procedure	Description
Step 1: Create an OBS Bucket Policy and ACL	Configure an OBS bucket policy and ACL using account A and grant account B required permissions like the read and write permissions.
Step 2: Create a Workload with an OBS Volume Mounted	Create a PV and a PVC based on the OBS bucket of account A using account B and mount the PVC to the required workload.
Step 3: Check the Pod Actions on the OBS Bucket	Check whether the pod created by account B has the required permissions based on the bucket policy.
Step 4: Clear Resources	Once you have studied this example, delete any associated resources to prevent incurring settlement fees.

Prerequisites

- The involved accounts are in the same region.
- You have created a cluster where the CCE Container Storage (Everest) add-on is installed. The add version must be 1.1.11 or later, and the cluster version must be 1.15 or later. If no cluster is available, create one by referring to [Buying a CCE Standard/Turbo Cluster](#).
- An ECS with an EIP bound has been created in the same VPC as the cluster, and the ECS has been connected to the cluster through kubectl. For details

about how to connect an ECS to a cluster, see [Connecting to a Cluster Using kubectl](#).

Step 1: Create an OBS Bucket Policy and ACL

Configure an OBS bucket policy and ACL using account A and grant account B required permissions like the read and write permissions.

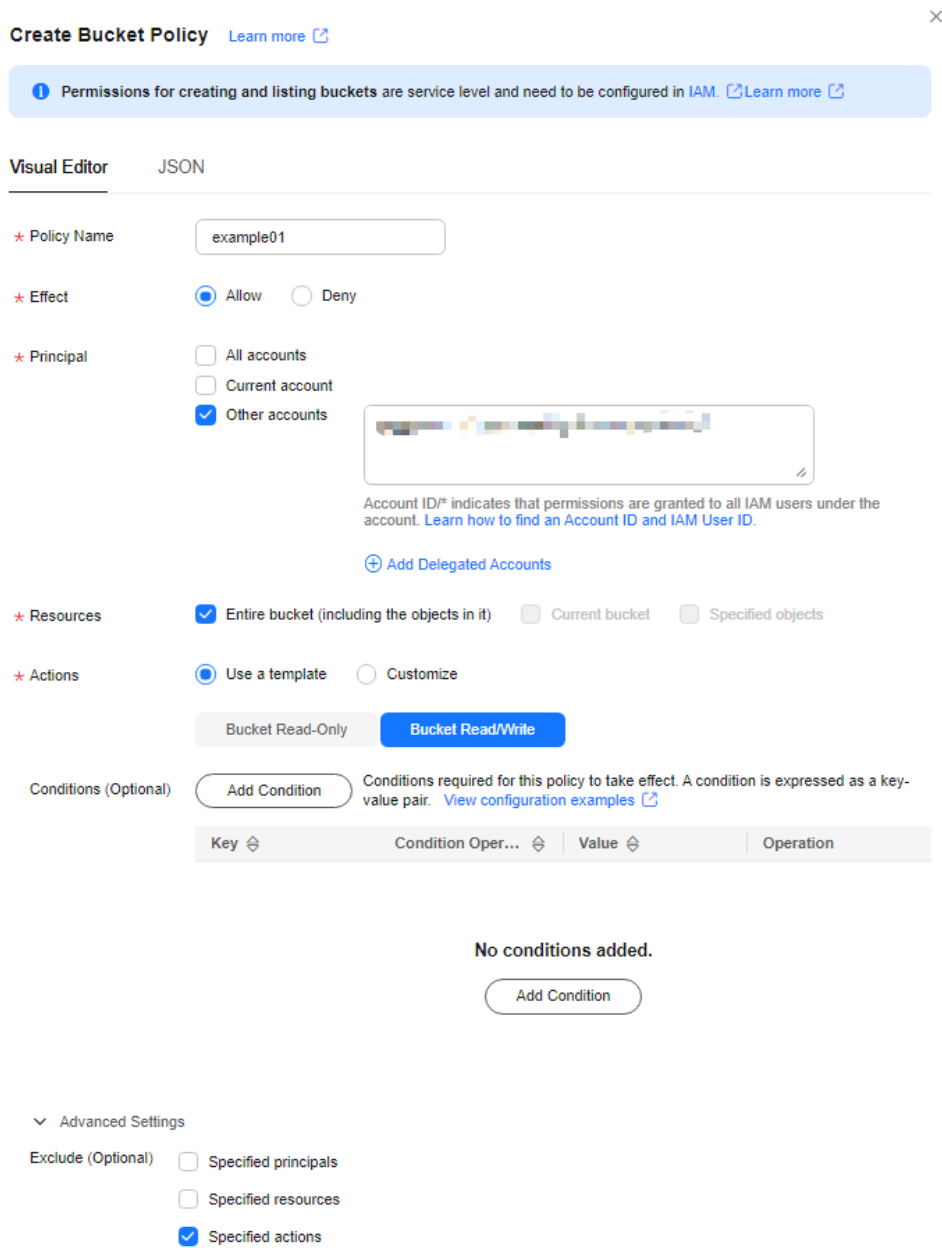
- Step 1** Log in to the OBS console. In the navigation pane, choose **Object Storage**.
- Step 2** Click the name of the target bucket to go to the **Objects** page.
- Step 3** In the navigation pane, choose **Permissions > Bucket Policies**. On the page displayed, click **Create**.
- Step 4** Configure the parameters. In this example, only some mandatory parameters are described. You can keep the default values for other parameters. For details about the parameters, see [Bucket Policies](#).

Table 10-3 Bucket policy parameters

Parameter	Description	Example
Policy Name	Enter a name.	example01
Effect	Specify the behavior of a policy. <ul style="list-style-type: none"> • Allow: The actions defined in the policy are allowed. • Deny: The actions defined in the policy are denied. 	Allow
Principal	Specify authorized accounts. (Multiple accounts can be selected.) For different types of authorized accounts, the OBS console provides different templates for authorizations. For details, see Creating a Bucket Policy with a Template . <ul style="list-style-type: none"> • All accounts: Any account can execute the current bucket policy without identity authentication, which may pose data security risks. • Current account: Grant permissions to a specific IAM account under the current account. • Other accounts: Grant permissions to a specific IAM account under another account. 	Other accounts XXX(account ID)/XXX (IAM ID)

Parameter	Description	Example
Resources	Specify the authorized resources. <ul style="list-style-type: none"> ● Entire bucket (including the objects in it): Allow authorized accounts to perform certain actions on a bucket and the objects in it. ● Current bucket: Allow authorized accounts to perform certain actions on the current bucket. ● Specific objects: Allow authorized accounts to perform certain actions on the specified objects in a bucket. 	Entire bucket (including the objects in it)
Actions	Specify actions. <ul style="list-style-type: none"> ● Use a template: Use a permission template preset on the OBS console. If you selected Bucket Read/Write, the Specified actions option will be selected by default in the Advanced Settings area. ● Customize: Customize the actions. 	Use a template > Bucket Read/Write

Figure 10-2 Creating a bucket policy



Step 5 In the navigation pane, choose **Permissions > Bucket ACL**. In the right pane, click **Add** under **User Access**. Enter the account ID of the authorized user, select **Read** and **Write** for **Access to Bucket**, select **Read** for **Access to Objects**, select **Read** and **Write** for **Access to ACL**, and click **OK**.

----End

Step 2: Create a Workload with an OBS Volume Mounted

Create a PV and a PVC based on the OBS bucket of account A using account B and mount the PVC to the required workload.

Step 1 Create a ConfigMap named **paas-obs-endpoint** and configure the region and endpoint of OBS.

```
vim config.yaml
```

The content is as follows: (For details about the parameters, see [Table 10-4.](#))

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: paas-obs-endpoint # The value must be paas-obs-endpoint.
  namespace: kube-system # The value must be kube-system.
data:
  obs-endpoint: |
    {"<region_name>": "<endpoint_address>"}
```

Create the ConfigMap using **config.yaml**.

```
kubectl create -f config.yaml
```

Table 10-4 ConfigMap parameters

ConfigMap	Description
metadata.name	ConfigMap name, which is fixed at paas-obs-endpoint and cannot be changed
metadata.namespace	Namespace, which is fixed at kube-system and cannot be changed
data.obs-endpoint	Region names and endpoints are in key-value pairs. Replace <i><region_name></i> and <i><endpoint_address></i> with specific values. If multiple values are needed, use commas (,) to separate them.

Step 2 Create a secret named **test-user**. (This secret is used to provide access credentials when volumes are mounted to CSI, and its name can be customized.)

1. Obtain the AK. Go back to the management console, hover the cursor over the username in the upper right corner and choose **My Credentials** from the drop-down list.

In the navigation pane, choose **Access Keys**. On the page displayed, click **Create Access Key**.

Click **OK** and download the AK.

2. Encode the AK using Base64 and save the encoded AK and SK. If the AK obtained is **xxx** and the SK is **yyy**, run the following commands:

```
echo -n xxx|base64
echo -n yyy|base64
```

3. Create a secret YAML file, for example, **test_user.yaml**.

```
vim test_user.yaml
```

The content is as follows: (For details about the parameters, see [Table 10-5.](#))

```
apiVersion: v1
data:
  access.key: QUxPQUUJ*****
  secret.key: aVMwZkduQ*****
kind: Secret
metadata:
  name: test-user
  namespace: default
type: cfe/secure-opaque
```

Create a secret using **test_user.yaml**.

```
kubectl create -f test_user.yaml
```

Table 10-5 Secret parameters

Parameter	Description	Example
access.key	A Base64-encoded AK	QUxPQUJU*****
secret.key	A Base64-encoded SK	aVMwZkduQ*****
type	Key type, which is fixed at cfe/secure-opaque and cannot be changed When this type is used, the data entered by users will be automatically encrypted.	cfe/secure-opaque

Step 3 Create a PV named **testing_abc** and mount the secret named **test_user** to the PV.

```
vim testing_abc.yaml
```

The content is as follows: (For details about the parameters, see [Table 10-6](#).)

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: testing-abc
  annotations:
    pv.kubernetes.io/bound-by-controller: 'yes'
    pv.kubernetes.io/provisioned-by: everest-csi-provisioner
spec:
  capacity:
    storage: 1Gi
  mountOptions:
    - default_acl=bucket-owner-full-control #New OBS mounting parameters
  csi:
    driver: obs.csi.everest.io
    volumeHandle: obs-cce-test # Name of the OBS bucket to be mounted
    fsType: s3fs # obsfs indicates a parallel file system, and s3fs indicates an OBS bucket.
    volumeAttributes:
      everest.io/obs-volume-type: STANDARD # Bucket type, which can be STANDARD or WARM when an
      OBS bucket is used
      everest.io/region: <region_name> # Region where the OBS bucket is located (Replace it with the
      actual value.)
    storage.kubernetes.io/csiProvisionerIdentity: everest-csi-provisioner
    nodePublishSecretRef: # AK/SK used for mounting the OBS bucket
      name: test-user
      namespace: default
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain # PV reclaim policy
  storageClassName: csi-obs # csi-obs specifies an OBS storage class that is automatically
  created. You can customize it as required.
  volumeMode: Filesystem
```

Create the PV using **testing_abc.yaml**.

```
kubectl create -f testing_abc.yaml
```


Table 10-6 PV parameters

Parameter	Description	Example
mountOptions.default_acl	<p>Specify access control policies for a bucket and objects in the bucket. In this example, account A owns the bucket, and both account A and account B have the ability to upload data.</p> <ul style="list-style-type: none"> • private: The bucket or objects can only be fully accessed by the owner of the bucket. • public-read: The owner of the bucket has complete control over both the bucket and its objects. While other users can read data from the bucket, they are unable to modify, delete, or upload any data within it. • public-read-write: The owner of the bucket has complete control over the bucket and its objects. Other users can read and write data from and to the bucket. • bucket-owner-read: Users who have uploaded objects to the bucket has complete control over the objects, while the bucket owner is only granted read permissions for said objects. This mode is usually used in cross-account sharing scenarios. • bucket-owner-full-control: Users who have uploaded objects to the bucket are granted write permissions for those specific objects, but not read permissions by default. The bucket owner has complete control over all objects within the bucket. This mode is usually used in cross-account sharing scenarios. 	<p>bucket-owner-full-control</p> <p>NOTE Due to the bucket policy that was configured using account A, account B has been granted read and write permissions for the entire bucket, including objects uploaded by both account A and B.</p>
csi.nodePublishSecretRef	<p>Specify the secret to be mounted.</p> <ul style="list-style-type: none"> • name: name of a secret • namespace: namespace where the secret is in 	<p>test-user default</p>
csi.volumeHandle	<p>Specify the name of the OBS bucket to be mounted.</p>	<p>obs-cce-test (OBS bucket name authorized by account A)</p>

Parameter	Description	Example
csi.fsType	Specify the file type. <ul style="list-style-type: none"> • obsfs: Create an OBS parallel file system. • s3fs: Create an OBS bucket. 	s3fs NOTICE The file type specified by the PV and the PVC must match in order for the PV to be bound to the corresponding PVC. If they do not match, the binding cannot occur.
accessModes	Specify the access mode of the storage volume. OBS supports only ReadWriteMany . <ul style="list-style-type: none"> • ReadWriteOnce: A storage volume can be mounted to a single node in read-write mode. • ReadWriteMany: A storage volume can be mounted to multiple nodes in read-write mode. 	ReadWriteMany
persistentVolumeReclaimPolicy	Specify the reclaim policy for the PV. <ul style="list-style-type: none"> • Delete: When a PVC is deleted, both the PV and underlying storage resources will be deleted. If the everest.io/reclaim-policy: retain-volume-only annotation is added to the YAML file, the underlying storage resources will be retained. • Retain: When a PVC is deleted, both the PV and underlying storage resources will be retained. You need to manually delete them. After the PVC is deleted, the PV is in the Released state and cannot be bound to a PVC again. 	Retain NOTE If multiple PVs use the same OBS volume, use Retain to prevent the underlying volume from being deleted with one of the PV.

Step 4 Create a PVC named **pvc-test-abc** and bind the new PV **testing_abc** to it.

```
vim pvc_test_abc.yaml
```

The file content is as follows:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-test-abc
  namespace: default
annotations:
  csi.storage.k8s.io/node-publish-secret-name: test-user # Mount a secret.
  csi.storage.k8s.io/node-publish-secret-namespace: default # Namespace of the secret
  everest.io/obs-volume-type: STANDARD # Bucket type, which can be STANDARD or WARM when an
OBS bucket is used
  csi.storage.k8s.io/fstype: s3fs # File type. obsfs indicates a parallel file system, and s3fs indicates
an OBS bucket.
  volume.beta.kubernetes.io/storage-provisioner: everest-csi-provisioner
spec:
```

```
accessModes:
- ReadWriteMany      # The value must be ReadWriteMany for object storage.
resources:
requests:
  storage: 1Gi      # Storage capacity of a PVC. This parameter is valid only for verification (fixed to 1,
cannot be empty or 0). The value setting does not take effect for OBS buckets.
  storageClassName: csi-obs # csi-obs specifies an OBS storage class that is automatically created. You can
customize it as required.
  volumeName: testing-abc # PV name
```

Create the PVC using **pvc_test_abc.yaml**.

```
kubectl create -f pvc_test_abc.yaml
```

Step 5 Create a workload and mount the PVC to it. The following uses an Nginx Deployment as an example.

```
vim obs_deployment_example.yaml
```

The file content is as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: obs-deployment-example      # Workload name, which can be customized
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: obs-deployment-example    # Label, which can be customized
  template:
    metadata:
      labels:
        app: obs-deployment-example
    spec:
      containers:
      - image: nginx
        name: container-0
        volumeMounts:
        - mountPath: /tmp            # PVC mount path, which can be customized as required
          name: pvc-obs-example
      restartPolicy: Always
      imagePullSecrets:
      - name: default-secret
      volumes:
      - name: pvc-obs-example
        persistentVolumeClaim:
          claimName: pvc-test-abc    # PVC name
```

Create the workload named **obs-deployment-example** using **obs_deployment_example.yaml**.

```
kubectl create -f obs_deployment_example.yaml
```

Check whether the workload has been created.

```
kubectl get pod
```

If information similar to the following is displayed and the workload is in the **Running** state, the workload has been created.

NAME	READY	STATUS	RESTARTS	AGE
obs-deployment-example-6b4dfd7b57-frfxv	1/1	Running	0	22h

----End

Step 3: Check the Pod Actions on the OBS Bucket

Check whether the pod created by account B has the required permissions based on the bucket policy.

- Step 1** Check whether the pod can read and write objects in the OBS bucket created by account A and assume that a **test.txt** file is present in the OBS bucket.

Run the following command to access the created workload. (You can press **Ctrl +D** to exit the current workload.)

```
kubectrl -n default exec -it obs-deployment-example-6b4dfd7b57-frfxv -c container-0 /bin/bash
```

Run the following command to check the pod actions on **test.txt**. **/tmp** specifies the PVC mount path.

```
ls -l /tmp/test.txt
```

If information similar to the following is displayed, the pod has the read and write permissions on the **test.txt** file, which is related to the bucket policy set by account A.

```
-rwxrwxrwx 1 root root 4 Sep  5 09:09 /tmp/test.txt
```

- Step 2** Check whether the pod can read and write data from and to the objects uploaded by itself in the OBS bucket.

Create a **test01.txt** file in **/tmp** and write **test\n** into the file.

```
echo -e "test\n" > /tmp/test01.txt
```

Run the following command to check the **test01.txt** content and check whether the pod can read and write new objects uploaded by itself: (Account A can check the new objects in the OBS bucket.)

```
cat /tmp/test01.txt
```

If information similar to the following is displayed, the pod has the read and write permissions on the objects uploaded by itself.

```
test
```

----End

Step 4: Clear Resources

Once you have studied this example, delete any associated resources to prevent incurring settlement fees. If you plan to learn other examples, wait until they are finished before doing any clean-up.

- Step 1** Run the following command to delete the workload:

```
kubectrl delete -f obs_deployment_example.yaml
```

Information similar to the following is displayed:

```
deployment.apps "obs-deployment-example" deleted
```

- Step 2** Run the following command to delete the PVC:

```
kubectrl delete -f pvc_test_abc.yaml
```

Information similar to the following is displayed:

```
persistentvolumeclaim "pvc-test-abc" deleted
```

Step 3 Run the following command to delete the PV:

```
kubectl delete -f testing_abc.yaml
```

Information similar to the following is displayed:

```
persistentvolume "testing-abc" deleted
```

Step 4 Run the following command to delete the secret:

```
kubectl delete -f test_user.yaml
```

Information similar to the following is displayed:

```
secret "test-user" deleted
```

Step 5 Run the following command to delete the ConfigMap:

```
kubectl delete -f config.yaml
```

Information similar to the following is displayed:

```
configmap "paas-obs-endpoint" deleted
```

----End

Common Issues

If a workload fails to be created, locate the fault based on the error information in the pod events. For details, see [Table 10-7](#).

Table 10-7 Locating the fault

Error	Possible Cause	Fault Locating
0/4 nodes are available: pod has unbound immediate PersistentVolumeClaims. preemption: 0/4 nodes are available: 4 Preemption is not helpful for scheduling.	The PVC is not bound to any PV.	<ol style="list-style-type: none"> Run the following command to check the PVC status: <pre>kubectl get pv</pre> If the PVC is in the Pending state, it is not bound to any PV. Check the PVC details and locate the cause of the binding failure. <pre>kubectl describe pv <pv_name></pre> Modify the YAML file to rectify the fault if the fault is caused by any of the following reasons: <ul style="list-style-type: none"> The PVC is not bound to the proper PV. The PVC and PV parameters do not match. This includes fsType, StorageClass, accessModes, and storage. The StorageClass must be object storage, and accessModes must be set to ReadWriteMany because OBS buckets only support this mode. Additionally, the storage value requested by the PVC must be equal to or less than the storage value provided by the PV.

Error	Possible Cause	Fault Locating
<p>MountVolume.Setup failed for volume "obs-cce-example": rpc error: code = Unknown desc = failed to get secret(paas.longaksk), err: get secret(paas.longaksk) failed: get secret paas.longaksk from namespace kube-system failed: secrets "paas.longaksk" not found</p>	<p>The required secret cannot be found when the storage volume is mounted to the workload.</p>	<ol style="list-style-type: none"> 1. Check whether the mounted secret is present. <code>kubectl get secret</code> If it is present, the secret may be incorrectly configured. If it is not present, you can create one by referring to Step 2. 2. Check the secret parameter configurations. The following shows the common issues: <ul style="list-style-type: none"> • The access.key and secret.key parameters are not set to the AK and SK encoded using Base64. • The type parameter is not set to cfe/secure-opaque.
<p>MountVolume.Setup failed for volume "pv-obs-example": rpc error: code = Internal desc = [8032c354-4e1b-41b0-81ce-9d4b3f8c49c9] get obsUrl failed before mount bucket obs-cce-example, get configMap paas-obs-endpoint from namespace kube-system failed: configmaps "paas-obs-endpoint" not found</p>	<p>The ConfigMap that stores the OBS endpoint is not present or is incorrectly configured.</p>	<ol style="list-style-type: none"> 1. Check whether the ConfigMap is present. <code>kubectl get configmap</code> If it is present, the ConfigMap may be incorrectly configured. If it is not present, you can create one by referring to Step 1. 2. Check the ConfigMap parameter configurations. The following shows the common issues: <ul style="list-style-type: none"> • The name parameter is not set to paas-obs-endpoint. • The namespace parameter is not set to kube-system. • The region name is not set to the region where the OBS bucket is located.

10.3 Dynamically Creating an SFS Turbo Subdirectory Using StorageClass

Background

The minimum capacity of an SFS Turbo file system is 500 GiB, and the SFS Turbo file system cannot be billed by usage. By default, the root directory of an SFS Turbo file system is mounted to a container which, in most case, does not require such a large capacity.

The everest add-on allows you to dynamically create subdirectories in an SFS Turbo file system and mount these subdirectories to containers. In this way, an SFS Turbo file system can be shared by multiple containers to increase storage efficiency.

Constraints

- Only clusters of v1.15 or later are supported.
- The cluster must use the everest add-on of version 1.1.13 or later.
- Kata containers are not supported.
- When the everest add-on earlier than 1.2.69 or 2.1.11 is used, a maximum of 10 PVCs can be created concurrently at a time by using the subdirectory function. everest of 1.2.69 or later or of 2.1.11 or later is recommended.
- A subPath volume is a subdirectory of an SFS Turbo file system. Increasing the capacity of a PVC of this type only changes the resource range specified by the PVC, but does not change the total capacity of the SFS Turbo file system. If the SFS Turbo file system's total resource capacity is not enough, the available capacity of the subPath volume will be restricted. To fix this, you must increase the resource capacity of the SFS Turbo file system on the SFS Turbo console.

Deleting the subPath volume does not result in the deletion of the resources of the SFS Turbo file system.

Creating an SFS Turbo Volume of the subPath Type

Step 1 Create an SFS Turbo file system in the same VPC and subnet as the cluster.

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

The following is an example:

```
apiVersion: storage.k8s.io/v1
allowVolumeExpansion: true
kind: StorageClass
metadata:
  name: sfsturbo-subpath-sc
mountOptions:
- lock
parameters:
  csi.storage.k8s.io/csi-driver-name: sfsturbo.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/archive-on-delete: "true"
  everest.io/share-access-to: 7ca2dba2-1234-1234-1234-626371a8fb3a
```

```

everest.io/share-expand-type: bandwidth
everest.io/share-export-location: 192.168.1.1:/sfsturbo/
everest.io/share-source: sfs-turbo
everest.io/share-volume-type: STANDARD
everest.io/volume-as: subpath
everest.io/volume-id: 0d773f2e-1234-1234-1234-de6a35074696
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
    
```

In this example:

- **name**: indicates the name of the StorageClass.
- **mountOptions**: indicates the mount options. This field is optional.
 - In versions later than everest 1.1.13 and earlier than everest 1.2.8, only the **noLock** parameter can be configured. By default, the **noLock** parameter is used for the mount operation and does not need to be configured. If **noLock** is set to **false**, the **lock** field is used.
 - Starting from everest 1.2.8, more mount options are supported. For details, see [Configuring SFS Volume Mount Options](#). **Do not set noLock to true. Otherwise, the mount operation will fail.**

```

mountOptions:
- vers=3
- timeo=600
- noLock
- hard
    
```

- **everest.io/volume-as**: This parameter is set to **subpath** to use the subPath volume.
- **everest.io/share-access-to**: This parameter is optional. In a subPath volume, set this parameter to the ID of the VPC where the SFS Turbo file system is located.
- **everest.io/share-expand-type**: This parameter is optional. If the type of the SFS Turbo file system is SFS Turbo Standard – Enhanced or SFS Turbo Performance – Enhanced, set this parameter to **bandwidth**.
- **everest.io/share-export-location**: This parameter indicates the mount directory. It consists of the SFS Turbo shared path and sub-directory. The shared path can be obtained on the SFS Turbo console. The sub-directory is user-defined. The PVCs created using the StorageClass are located in this sub-directory.
- **everest.io/share-volume-type**: This parameter is optional. It specifies the SFS Turbo file system type. The value can be **STANDARD** or **PERFORMANCE**. For enhanced types, this parameter must be used together with **everest.io/share-expand-type** (whose value should be **bandwidth**).
- **everest.io/zone**: This parameter is optional. Set it to the AZ where the SFS Turbo file system is located.
- **everest.io/volume-id**: This parameter indicates the ID of the SFS Turbo volume. You can obtain the volume ID on the SFS Turbo page.
- **everest.io/archive-on-delete**: If this parameter is set to **true** and **Delete** is selected for **Reclaim Policy**, the original documents of the PV will be archived to the directory named **archived-*{PV name.timestamp}*** before the PVC is deleted. If this parameter is set to **false**, the SFS Turbo subdirectory of the corresponding PV will be deleted. The default value is **true**, indicating that the original documents of the PV will be archived to the directory named **archived-*{PV name.timestamp}*** before the PVC is deleted.

Step 3 Run `kubectl create -f sfsturbo-subpath-sc.yaml`.

Step 4 Create a PVC YAML file named `sfs-turbo-test.yaml`.

The following is an example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: sfs-turbo-test
  namespace: default
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
  storageClassName: sfsturbo-subpath-sc
  volumeMode: Filesystem
```

In this example:

- **name:** indicates the name of the PVC.
- **storageClassName:** specifies the name of the StorageClass.
- **storage:** In a subPath volume, modifying the value of this parameter does not impact the resource capacity of the SFS Turbo file system. A subPath volume is essentially a file path within an SFS Turbo file system. As a result, increasing the capacity of the subPath volume in a PVC does not lead to an increase in the resources of the SFS Turbo file system.

NOTE

The capacity of a subPath volume is restricted by the overall resource capacity of the corresponding SFS Turbo file system. If the resources of the SFS Turbo file system are inadequate, you can adjust the resource capacity via the SFS Turbo console.

Step 5 Run `kubectl create -f sfs-turbo-test.yaml`.

----End

Creating a Deployment and Mounting an Existing Volume

Step 1 Create a YAML file for the Deployment, for example, `deployment-test.yaml`.

The following is an example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-turbo-subpath-example
  namespace: default
  generation: 1
  labels:
    appgroup: ""
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-turbo-subpath-example
  template:
    metadata:
      labels:
        app: test-turbo-subpath-example
    spec:
      containers:
```

```
- image: nginx:latest
  name: container-0
  volumeMounts:
  - mountPath: /tmp
    name: pvc-sfs-turbo-example
restartPolicy: Always
imagePullSecrets:
- name: default-secret
volumes:
- name: pvc-sfs-turbo-example
  persistentVolumeClaim:
    claimName: sfs-turbo-test
```

In this example:

- **name**: indicates the name of the created workload.
- **image**: specifies the image used by the workload.
- **mountPath**: indicates the mount path of the container. In this example, the volume is mounted to the **/tmp** directory.
- **claimName**: indicates the name of an existing PVC.

Step 2 Create the Deployment.

```
kubectl create -f deployment-test.yaml
```

----End

Dynamically Creating a subPath Volume for a StatefulSet

Step 1 Create a YAML file for a StatefulSet, for example, **statefulset-test.yaml**.

The following is an example:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: test-turbo-subpath
  namespace: default
  generation: 1
  labels:
    appgroup: ""
spec:
  replicas: 2
  selector:
    matchLabels:
      app: test-turbo-subpath
  template:
    metadata:
      labels:
        app: test-turbo-subpath
    annotations:
      metrics.alpha.kubernetes.io/custom-endpoints: '[{"api":"","path":"","port":"","names":""}]'
      pod.alpha.kubernetes.io/initialized: 'true'
    spec:
      containers:
      - name: container-0
        image: 'nginx:latest'
        resources: {}
        volumeMounts:
        - name: sfs-turbo-160024548582479676
          mountPath: /tmp
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: IfNotPresent
        restartPolicy: Always
```

```
terminationGracePeriodSeconds: 30
dnsPolicy: ClusterFirst
securityContext: {}
imagePullSecrets:
  - name: default-secret
affinity: {}
schedulerName: default-scheduler
volumeClaimTemplates:
  - metadata:
      name: sfs-turbo-160024548582479676
      namespace: default
      annotations: {}
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 10Gi
      storageClassName: sfsturbo-subpath-sc
serviceName: wwwww
podManagementPolicy: OrderedReady
updateStrategy:
  type: RollingUpdate
revisionHistoryLimit: 10
```

In this example:

- **name**: indicates the name of the created workload.
- **image**: specifies the image used by the workload.
- **mountPath**: indicates the mount path of the container. In this example, the volume is mounted to the **/tmp** directory.
- **spec.template.spec.containers.volumeMounts.name** and **spec.volumeClaimTemplates.metadata.name**: must be consistent because they have a mapping relationship.
- **storageClassName**: specifies the name of an on-premises StorageClass.

Step 2 Create the StatefulSet.

```
kubectl create -f statefulset-test.yaml
```

```
----End
```

10.4 Using Custom Storage Classes

Background

When using storage resources in CCE, the most common method is to specify **storageClassName** to define the type of storage resources to be created when creating a PVC. The following configuration shows how to use a PVC to apply for an SAS (high I/O) EVS disk (block storage).

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
  namespace: default
  annotations:
    everest.io/disk-volume-type: SAS
spec:
  accessModes:
    - ReadWriteOnce
```

```
resources:
  requests:
    storage: 10Gi
  storageClassName: csi-disk
```

To specify the EVS disk type, you can configure the **everest.io/disk-volume-type** field. The value **SAS** is used as an example here, indicating the high I/O EVS disk type. Or you can choose **SSD** (ultra-high I/O).

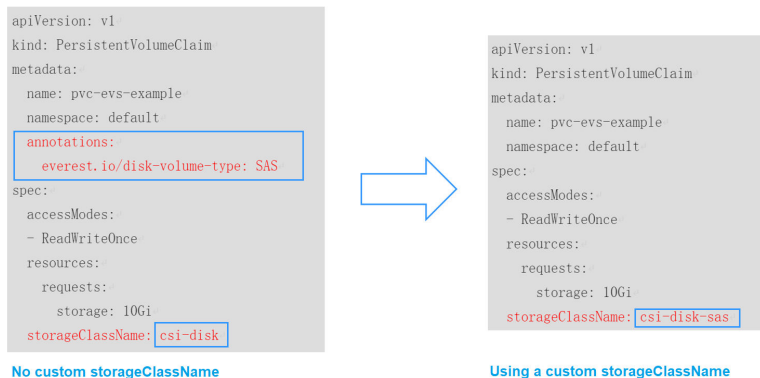
This configuration method may not work if you want to:

- Set **storageClassName** only, which is simpler than specifying the EVS disk type by using **everest.io/disk-volume-type**.
- Avoid modifying YAML files or Helm charts. Some users switch from self-built or other Kubernetes services to CCE and have written YAML files of many applications. In these YAML files, different types of storage resources are specified by different StorageClassNames. When using CCE, they need to modify a large number of YAML files or Helm charts to use storage resources, which is labor-consuming and error-prone.
- Set the default **storageClassName** for all applications to use the default storage class. In this way, you can create storage resources of the default type without needing to specify **storageClassName** in the YAML file.

Solution

This section describes how to set a custom storage class in CCE and how to set the default storage class. You can specify different types of storage resources by setting **storageClassName**.

- For the first scenario, you can define custom storageClassNames for SAS and SSD EVS disks. For example, define a storage class named **csi-disk-sas** for creating SAS disks. The following figure shows the differences before and after you use a custom storage class.



- For the second scenario, you can define a storage class with the same name as that in the existing YAML file without needing to modify **storageClassName** in the YAML file.
- For the third scenario, you can set the default storage class as described below to create storage resources without specifying **storageClassName** in YAML files.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-evs-example
```

```
namespace: default
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Creating a StorageClass Using a YAML File

As of now, CCE provides StorageClasses such as `csi-disk`, `csi-nas`, and `csi-obs` by default. When defining a PVC, you can use a **StorageClassName** to automatically create a PV of the corresponding type and automatically create underlying storage resources.

Run the following `kubectl` command to obtain the StorageClasses that CCE supports. Use the CSI add-on provided by CCE to create a StorageClass.

```
# kubectl get sc
NAME                PROVISIONER             AGE      #
csi-disk            everest-csi-provisioner 17d      # EVS disk
csi-disk-topology  everest-csi-provisioner 17d      # EVS disks created with delay
csi-nas             everest-csi-provisioner 17d      # SFS 1.0
csi-obs            everest-csi-provisioner 17d      # OBS
csi-sfsturbo      everest-csi-provisioner 17d      # SFS Turbo
```

Each StorageClass contains the default parameters used for dynamically creating a PV. The following is an example of StorageClass for EVS disks:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/passthrough: 'true'
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

Table 10-8 Key parameters

Parameter	Description
<code>provisioner</code>	Specifies the storage resource provider, which is the Everest add-on for CCE. Set this parameter to everest-csi-provisioner .
<code>parameters</code>	Specifies the storage parameters, which vary with storage types. For details, see Table 10-9 .

Parameter	Description
reclaimPolicy	<p>Specifies the value of persistentVolumeReclaimPolicy for creating a PV. The value can be Delete or Retain. If reclaimPolicy is not specified when a StorageClass object is created, the value defaults to Delete.</p> <ul style="list-style-type: none"> • Delete: When a PVC is deleted, its associated underlying storage resources will be deleted and the PV resources will be removed. Exercise caution if you select this option. • Retain: When a PVC is deleted, both of the PV and its associated underlying storage resources will be retained and the PV is marked as released. If you manually delete the PV afterwards, the underlying storage resources will not be deleted. To bind the PV to a new PVC, you need to remove the original binding information from the PV.
allowVolume Expansion	<p>Specifies whether the PV of this StorageClass supports dynamic capacity expansion. The default value is false. Dynamic capacity expansion is implemented by the underlying storage add-on. This is only a switch.</p>
volumeBindingMode	<p>Specifies the volume binding mode, that is, the time when a PV is dynamically created. The value can be Immediate or WaitForFirstConsumer.</p> <ul style="list-style-type: none"> • Immediate: After a PVC is created, the storage resources and PV will be created and associated with the PVC without delay. • WaitForFirstConsumer: After a PVC is created, it will not be immediately bound to a PV. Instead, the storage resources and PV will be generated and bound to the PVC only after the pod that requires the PVC is scheduled.
mountOptions	<p>This field must be supported by the underlying storage. If this field is not supported but is specified, the PV creation will fail.</p>

Table 10-9 Parameters

V o l u m e T y p e	Param eter	Ma n d a t o r y	Description	
E V S	csi.stor age.k8s .io/csi- driver- name	Yes	Driver type. If an EVS disk is used, the parameter value is fixed at disk.csi.everest.io .	
	csi.stor age.k8s .io/ fstype	Yes	If an EVS disk is used, the parameter value can be ext4 .	
	everest. io/disk- volume -type	Yes	EVS disk type. All letters are in uppercase. <ul style="list-style-type: none"> • SAS: high I/O • SSD: ultra-high I/O 	
	everest. io/ passth rough	Yes	The parameter value is fixed at true , which indicates that the EVS device type is SCSI . Other parameter values are not allowed.	
	SFS Turbo	csi. sto rag e.k 8s.i o/ csi- dri ver - na me	Yes	Driver type. If SFS Turbo is used, the parameter value is fixed at sfsturbo.csi.everest.io .

V o l u m e T y p e	Param eter	Ma n d a t o r y	Description	
		csi. sto rag e.k 8s.i o/ fst ype	Yes	If SFS Turbo is used, the value can be nfs .
ev er es t.i o/ sh ar e- ac ce ss - to		Yes	VPC ID of the cluster.	
ev er es t.i o/ sh ar e- ex p a n d- ty p e		No	Extension type. The default value is bandwidth , indicating an enhanced file system. This parameter does not take effect.	

V o l u m e T y p e	Param eter	Ma n d a t o r y	Description	
ev er es t.i o/ sh ar e- so ur ce		Yes	The parameter value is fixed at sfs-turbo .	
ev er es t.i o/ sh ar e- v ol u m e- ty p e		No	SFS Turbo StorageClass. The default value is STANDARD , indicating standard and standard enhanced editions. This parameter does not take effect.	
O B S	csi.stor age.k8s .io/csi- driver- name	Yes	Driver type. If OBS is used, the parameter value is fixed at obs.csi.everest.io .	
	csi.stor age.k8s .io/ fstype	Yes	Instance type, which can be obsfs or s3fs . <ul style="list-style-type: none"> ● obsfs: a parallel file system ● s3fs: object bucket 	

V o l u m e T y p e	Param eter	Ma n d a t o r y	Description
	everest.io/obs-volume-type	Yes	<p>OBS StorageClass.</p> <ul style="list-style-type: none"> If fsType is set to s3fs, STANDARD (standard bucket) and WARM (infrequent access bucket) are supported. This parameter is invalid when fsType is set to obsfs.

Custom Storage Classes

You can customize a high I/O storage class in a YAML file. For example, the name **csi-disk-sas** indicates that the disk type is SAS (high I/O).

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-sas # Name of the high I/O storage class, which can be customized.
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS # High I/O EVS disk type, which cannot be customized.
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true # true indicates that capacity expansion is allowed.

```

For an ultra-high I/O storage class, you can set the class name to **csi-disk-ssd** to create SSD EVS disk (ultra-high I/O).

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd # Name of the ultra-high I/O storage class, which can be customized.
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD # Ultra-high I/O EVS disk type, which cannot be customized.
  everest.io/passthrough: "true"
provisioner: everest-csi-provisioner

```

```
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

reclaimPolicy: indicates the recycling policies of the underlying cloud storage. The value can be **Delete** or **Retain**.

- **Delete:** When a PVC is deleted, both the PV and the EVS disk are deleted.
- **Retain:** When a PVC is deleted, the PV and underlying storage resources are not deleted. Instead, you must manually delete these resources. After that, the PV resource is in the **Released** state and cannot be bound to the PVC again.

If high data security is required, you are advised to select **Retain** to prevent data from being deleted by mistake.

After the definition is complete, run the **kubectl create** commands to create storage resources.

```
# kubectl create -f sas.yaml
storageclass.storage.k8s.io/csi-disk-sas created
# kubectl create -f ssd.yaml
storageclass.storage.k8s.io/csi-disk-ssd created
```

Query the storage class again. Two more types of storage classes are displayed in the command output, as shown below.

```
# kubectl get sc
NAME          PROVISIONER          AGE
csi-disk      everest-csi-provisioner 17d
csi-disk-sas  everest-csi-provisioner 2m28s
csi-disk-ssd  everest-csi-provisioner 16s
csi-disk-topology everest-csi-provisioner 17d
csi-nas       everest-csi-provisioner 17d
csi-obs       everest-csi-provisioner 17d
csi-sfsturbo  everest-csi-provisioner 17d
```

Other types of storage resources can be defined in the similar way. You can use **kubectl** to obtain the YAML file and modify it as required.

- **File storage**

```
# kubectl get sc csi-nas -oyaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-nas
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: nas.csi.everest.io
  csi.storage.k8s.io/fstype: nfs
  everest.io/share-access-level: rw
  everest.io/share-access-to: 5e3864c6-e78d-4d00-b6fd-de09d432c632 # ID of the VPC to which the cluster belongs
  everest.io/share-is-public: 'false'
  everest.io/zone: xxxxx # AZ
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

- **Object storage**

```
# kubectl get sc csi-obs -oyaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-obs
provisioner: everest-csi-provisioner
```

```
parameters:
  csi.storage.k8s.io/csi-driver-name: obs.csi.everest.io
  csi.storage.k8s.io/fstype: s3fs # Object storage type. s3fs indicates an object bucket, and obsfs
  indicates a parallel file system.
  everest.io/obs-volume-type: STANDARD # Storage class of the OBS bucket
  reclaimPolicy: Delete
  volumeBindingMode: Immediate
```

Specifying an Enterprise Project for Storage Classes

CCE allows you to specify an enterprise project when creating EVS disks and OBS PVCs. The created storage resources (EVS disks and OBS) belong to the specified enterprise project. **The enterprise project can be the enterprise project to which the cluster belongs or the default enterprise project.**

If you do not specify any enterprise project, the enterprise project in StorageClass is used by default. The created storage resources by using the csi-disk and csi-obs storage classes of CCE belong to the default enterprise project.

If you want the storage resources created from the storage classes to be in the same enterprise project as the cluster, you can customize a storage class and specify the enterprise project ID, as shown below.

NOTE

To use this function, the everest add-on must be upgraded to 1.2.33 or later.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: csi-disk-epid #Customize a storage class name.
provisioner: everest-csi-provisioner
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SAS
  everest.io/enterprise-project-id: 86bfc701-9d9e-4871-a318-6385aa368183 #Specify the enterprise project
  ID.
  everest.io/passthrough: 'true'
  reclaimPolicy: Delete
  allowVolumeExpansion: true
  volumeBindingMode: Immediate
```

Specifying a Default Storage Class

You can specify a storage class as the default class. In this way, if you do not specify **storageClassName** when creating a PVC, the PVC is created using the default storage class.

For example, to specify **csi-disk-ssd** as the default storage class, edit your YAML file as follows:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-disk-ssd
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" # Specifies the default storage class in a cluster. A
    cluster can have only one default storage class.
parameters:
  csi.storage.k8s.io/csi-driver-name: disk.csi.everest.io
  csi.storage.k8s.io/fstype: ext4
  everest.io/disk-volume-type: SSD
  everest.io/passthrough: "true"
```

```
provisioner: everest-csi-provisioner
reclaimPolicy: Delete
volumeBindingMode: Immediate
allowVolumeExpansion: true
```

Delete the created `csi-disk-ssd` disk, run the **kubect** `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.

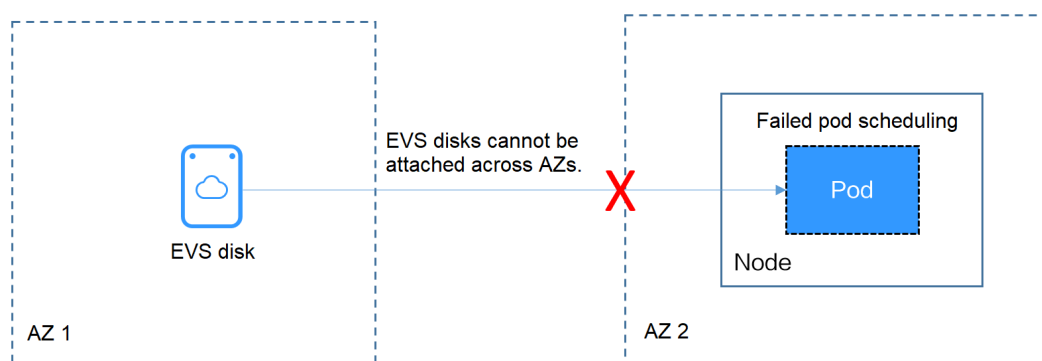
```
# kubectl create -f ssd-disk.yaml
persistentvolumeclaim/ssd-disk created
# kubectl get pvc
NAME      STATUS  VOLUME                                     CAPACITY  ACCESS MODES  STORAGECLASS  AGE
sas-disk  Bound   pvc-6e2f37f9-7346-4419-82f7-b42e79f7964c  10Gi      RWO           csi-disk-sas  16m
ssd-disk  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 Scheduling EVS Disks Across AZs Using `csi-disk-topology`

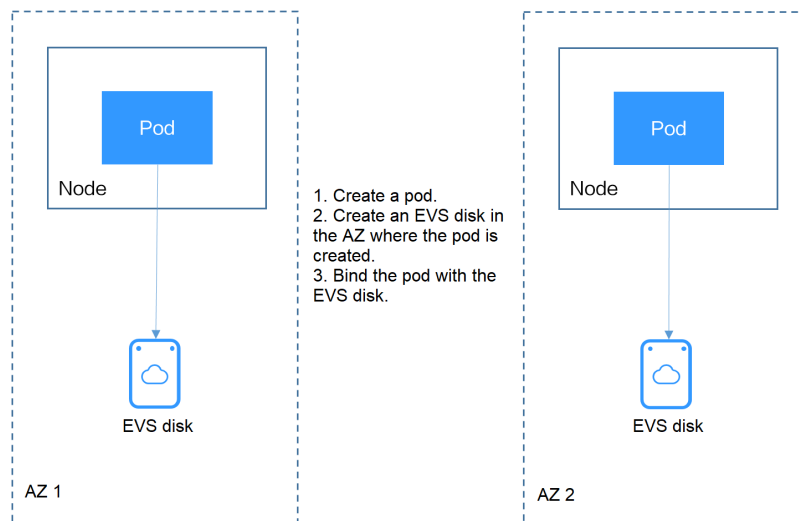
Background

EVS disks cannot be attached to a node deployed in another AZ. For example, the EVS disks in AZ 1 cannot be attached to a node in AZ 2. If the storage class `csi-disk` is used for StatefulSets, when a StatefulSet is scheduled, a PVC and a PV are created immediately (an EVS disk is created along with the PV), and then the PVC is bound to the PV. However, when the cluster nodes are located in multiple AZs, the EVS disk created by the PVC and the node to which the pods are scheduled may be in different AZs. As a result, the pods fail to be scheduled.



Solution

CCE provides a storage class named `csi-disk-topology`, which is a late-binding EVS disk type. When you use this storage class to create a PVC, no PV will be created in pace with the PVC. Instead, the PV is created in the AZ of the node where the pod will be scheduled. An EVS disk is then created in the same AZ to ensure that the EVS disk can be attached and the pod can be successfully scheduled.



Failed Pod Scheduling Due to csi-disk Used in Cross-AZ Node Deployment

Create a cluster with three nodes in different AZs.

Use the csi-disk storage class to create a StatefulSet and check whether the workload is successfully created.

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: nginx # Name of the headless Service
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: container-0
          image: nginx:alpine
          resources:
            limits:
              cpu: 600m
              memory: 200Mi
            requests:
              cpu: 600m
              memory: 200Mi
          volumeMounts: # Storage mounted to the pod
            - name: data
              mountPath: /usr/share/nginx/html # Mount the storage to /usr/share/nginx/html.
      imagePullSecrets:
        - name: default-secret
      volumeClaimTemplates:
        - metadata:
            name: data
            annotations:
              everest.io/disk-volume-type: SAS
          spec:
            accessModes:
              - ReadWriteOnce
            resources:

```

```
requests:
  storage: 1Gi
storageClassName: csi-disk
```

The StatefulSet uses the following headless Service.

```
apiVersion: v1
kind: Service      # Object type (Service)
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - name: nginx      # Name of the port for communication between pods
      port: 80        # Port number for communication between pods
  selector:
    app: nginx        # Select the pod whose label is app:nginx.
  clusterIP: None    # Set this parameter to None, indicating the headless Service.
```

After the creation, check the PVC and pod status. In the following output, the PVC has been created and bound successfully, and a pod is in the Pending state.

```
# kubectl get pvc -owide
NAME          STATUS  VOLUME                                     CAPACITY  ACCESS MODES  STORAGECLASS
AGE  VOLUMEMODE
data-nginx-0  Bound  pvc-04e25985-fc93-4254-92a1-1085ce19d31e  1Gi      RWO           csi-disk
64s  Filesystem
data-nginx-1  Bound  pvc-0ae6336b-a2ea-4ddc-8f63-cfc5f9efe189  1Gi      RWO           csi-disk
47s  Filesystem
data-nginx-2  Bound  pvc-aa46f452-cc5b-4dbd-825a-da68c858720d  1Gi      RWO           csi-disk
30s  Filesystem
data-nginx-3  Bound  pvc-3d60e532-ff31-42df-9e78-015cacb18a0b  1Gi      RWO           csi-disk
14s  Filesystem

# kubectl get pod -owide
NAME          READY  STATUS   RESTARTS  AGE  IP           NODE          NOMINATED NODE  READINESS GATES
nginx-0       1/1    Running  0          2m25s  172.16.0.12  192.168.0.121 <none>          <none>
nginx-1       1/1    Running  0          2m8s   172.16.0.136 192.168.0.211 <none>          <none>
nginx-2       1/1    Running  0          111s   172.16.1.7   192.168.0.240 <none>          <none>
nginx-3       0/1    Pending  0          95s    <none>       <none>        <none>          <none>
```

The event information of the pod shows that the scheduling fails due to no available node. Two nodes (in AZ 1 and AZ 2) do not have sufficient CPUs, and the created EVS disk is not in the AZ where the third node (in AZ 3) is located. As a result, the pod cannot use the EVS disk.

```
# kubectl describe pod nginx-3
Name:          nginx-3
...
Events:
  Type     Reason          Age    From          Message
  ----     -
Warning   FailedScheduling 111s   default-scheduler  0/3 nodes are available: 3 pod has unbound immediate PersistentVolumeClaims.
Warning   FailedScheduling 111s   default-scheduler  0/3 nodes are available: 3 pod has unbound immediate PersistentVolumeClaims.
Warning   FailedScheduling 28s    default-scheduler  0/3 nodes are available: 1 node(s) had volume node affinity conflict, 2 Insufficient cpu.
```

Check the AZ where the EVS disk created from the PVC is located. It is found that data-nginx-3 is in AZ 1. In this case, the node in AZ 1 has no resources, and only the node in AZ 3 has CPU resources. As a result, the scheduling fails. Therefore, there should be a delay between creating the PVC and binding the PV.

Storage Class for Delayed Binding

If you check the cluster storage class, you can see that the binding mode of `csi-disk-topology` is **WaitForFirstConsumer**, indicating that a PV is created and bound when a pod uses the PVC. That is, the PV and the underlying storage resources are created based on the pod information.

```
# kubectl get storageclass
NAME          PROVISIONER          RECLAIMPOLICY  VOLUMEBINDINGMODE  ALLOWVOLUMEEXPANSION  AGE
csi-disk      everest-csi-provisioner  Delete         Immediate           true                  156m
csi-disk-topology everest-csi-provisioner  Delete         WaitForFirstConsumer true                  156m
csi-nas       everest-csi-provisioner  Delete         Immediate           true                  156m
csi-obs       everest-csi-provisioner  Delete         Immediate           false                 156m
```

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 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	<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

An init container is a type of container that starts and exits before the application containers start. If there are multiple init containers, they will be started in the defined sequence. The data generated in the init containers can be used by the application containers because storage volumes in a pod are shared.

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 service pod is deployed. After the preparations are complete, the init container runs to completion and exits, 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, a pod is registered with a central database and application dependencies are downloaded.

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.

```

9dc822969e3f      percona          "docker-entrypoint..." 34 seconds ago    Up 33 seconds
ql_mysql-76598b8c64-mm9w 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-mm9w default_522566ea-bda5-11e9-a219-fa163e8b288b_0
615db9e60a80      cfe-pause:11.23.1 "/pause"                About a minute ago  Up About a minute
mysql-76598b8c64-mm9w default_522566ea-bda5-11e9-a219-fa163e8b288b_0

```

----End

11.4 Configuring the /etc/hosts File of a Pod Using hostAliases

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 Locating Container Faults Using the Core Dump File

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:

NOTE

In this example, the permission rules include the read-only permission (get/list/watch) of pods in the **test** namespace, and the read (get/list/watch) and create permissions of deployments.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: myrole
  namespace: test
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - apps
  resources:
  - pods
  - deployments
  verbs:
  - get
  - list
  - watch
  - create
```

Create a Role.

```
kubectl create -f role-test.yaml
```

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

Step 3 Create a RoleBinding and bind the service account to the role so that the user can obtain the corresponding permissions.

```
vi myrolebinding.yaml
```

The content is as follows:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: myrolebinding
  namespace: test
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: myrole
subjects:
- kind: ServiceAccount
  name: my-sa
  namespace: test
```

Create a RoleBinding.

```
kubectl create -f myrolebinding.yaml
```

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

The user information is configured. Now perform [Step 5](#) to [Step 7](#) to write the user information to the configuration file.

Step 4 Manually create a token that is valid for a long time for ServiceAccount.

```
vi my-sa-token.yaml
```

The content is as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-sa-token-secret
  namespace: test
  annotations:
    kubernetes.io/service-account.name: my-sa
type: kubernetes.io/service-account-token
```

Create a token:

```
kubectl create -f my-sa-token.yaml
```

Step 5 Configure the cluster information.

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

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

2. Set a cluster access mode. **test-arm** specifies the cluster to be accessed.

https://192.168.0.110:5443 specifies the apiserver IP address of the cluster. /
home/test.config specifies the path for storing the configuration file.

- If the internal API server address is used, run the following command:

```
kubectl config set-cluster test-arm --server=https://192.168.0.110:5443 --certificate-authority=/home/ca.crt --embed-certs=true --kubeconfig=/home/test.config
```
- If the public API server address is used, run the following command:

```
kubectl config set-cluster test-arm --server=https://192.168.0.110:5443 --kubeconfig=/home/test.config --insecure-skip-tls-verify=true
```

```
[root@test-arm-54016 home]# kubectl config set-cluster test-arm --server=https://10.0.1.100:5443 --certificate-authority=/home/ca.crt --embed-certs=true --kubeconfig=/home/test.config
Cluster "test-arm" set.
[root@test-arm-54016 home]# _
```

NOTE

If you perform operations on a node in the cluster or the node that uses the configuration is a cluster node, do not set the path of kubeconfig to **/root/.kube/config**.

By default, the apiserver IP address of the cluster is a private IP address. After an EIP is bound, you can use the public network IP address to access the apiserver.

Step 6 Configure the cluster authentication information.

1. Obtain the cluster token. (If the token is obtained in GET mode, run **base64 -d** to decode the token.)

```
token=$(kubectl describe secret my-sa-token-secret -n test | awk '/token:/{print $2}')
```

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

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

```
[root@test-arm-54016 home]# kubectl config set-credentials ui-admin --token=$token --kubeconfig=/home/test.config
User "ui-admin" set.
[root@test-arm-54016 home]#
```


Step 7 Configure the context information for cluster authentication access. **ui-admin@test** specifies the context name.

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

```
[root@test-arm-54016 home]# kubectl config set-context ui-admin@test --cluster=test-arm --user=ui-admin --kubeconfig=/home/test.config
Context "ui-admin@test" created.
[root@test-arm-54016 home]#
```

Step 8 Configure the context. For details about how to use the context, see [Verification](#).

```
kubectl config use-context ui-admin@test --kubeconfig=/home/test.config
```

```
[paas@test-arm-54016 home]# kubectl config use-context ui-admin@test --kubeconfig=/home/test.config
Switched to context "ui-admin@test".
[paas@test-arm-54016 home]#
```

NOTE

If you want to assign other users the above permissions to perform operations on the cluster, provide the generated configuration file **/home/test.config** to the user after performing step [Step 7](#). The user must ensure that the host can access the API server address of the cluster. When performing step [Step 8](#) on the host and using kubectl, the user must set the kubeconfig parameter to the path of the configuration file.

----End

Verification

1. Pods in the **test** namespace cannot access pods in other namespaces.

```
kubectl get pod -n test --kubeconfig=/home/test.config
```

```
[paas@test-arm-54016 home]# kubectl get pod -n test --kubeconfig=/home/test.config
NAME                READY   STATUS              RESTARTS   AGE
test-pod-56cfcbf45b-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 group "" 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 delete 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.

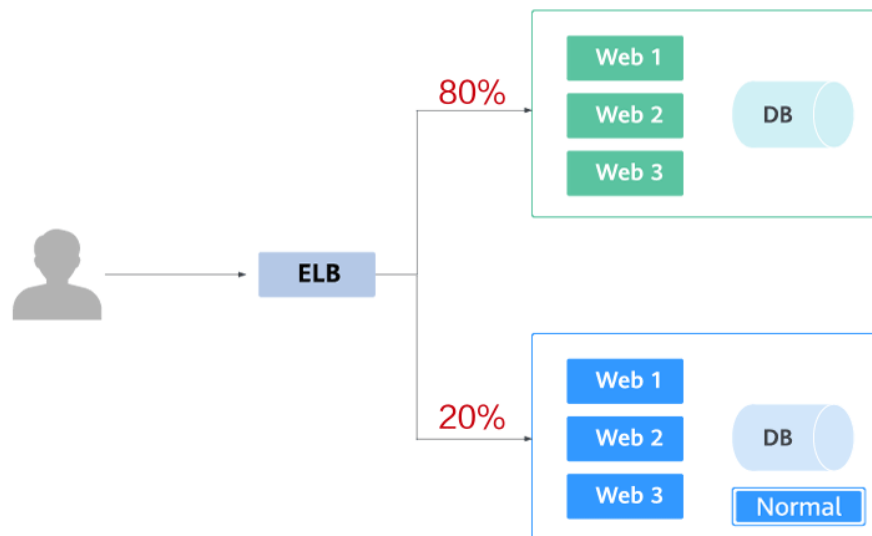
Solution

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.

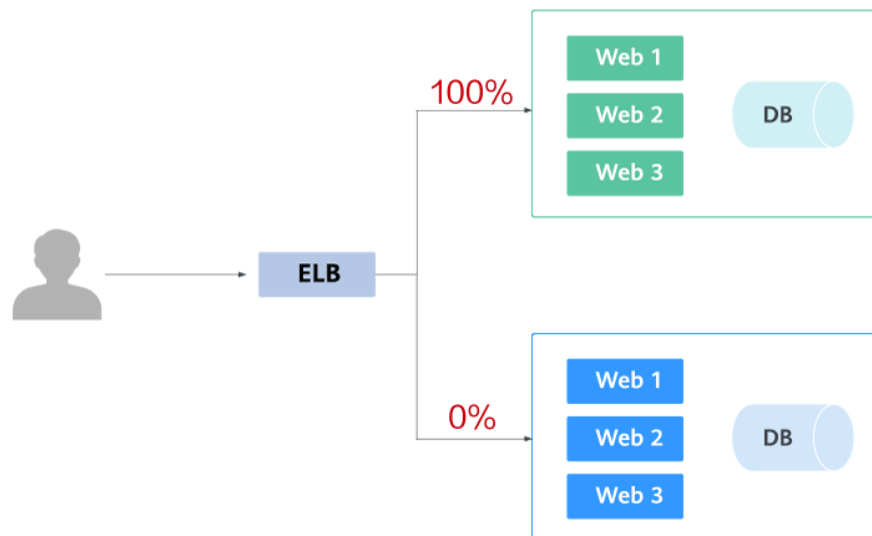
This document describes the principles and practices of grayscale release and blue-green deployment.

- 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.

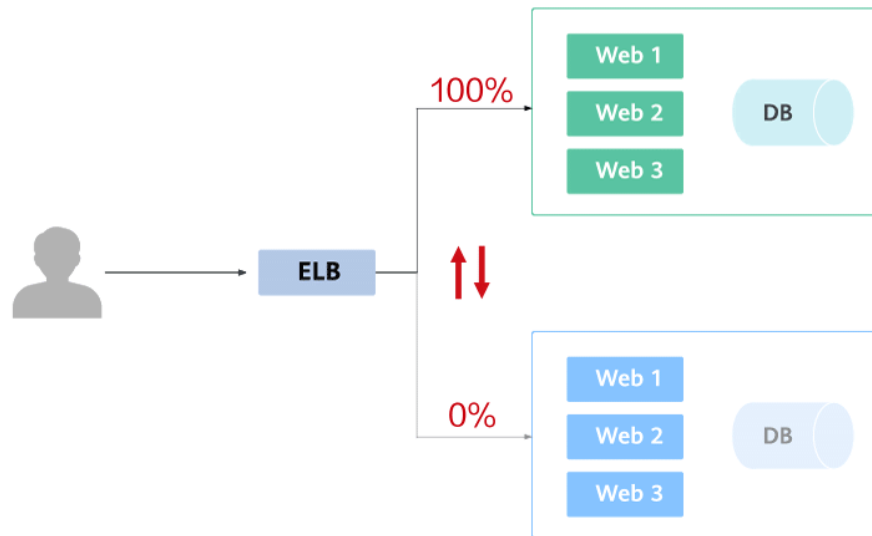
The following figure shows the general process of grayscale release. First, divide 20% of all service traffic to the new version. If the service version runs normally, gradually increase the traffic proportion and continue to test the performance of the new version. If the new version is stable, switch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

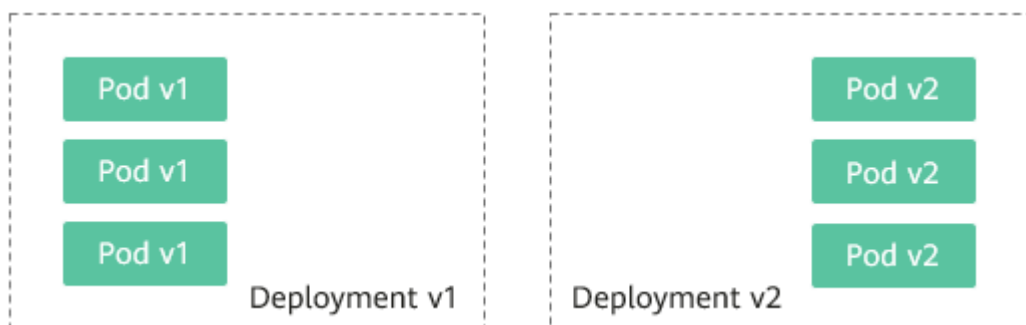


13.2 Using Services to Implement Simple Grayscale Release and Blue-Green Deployment

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

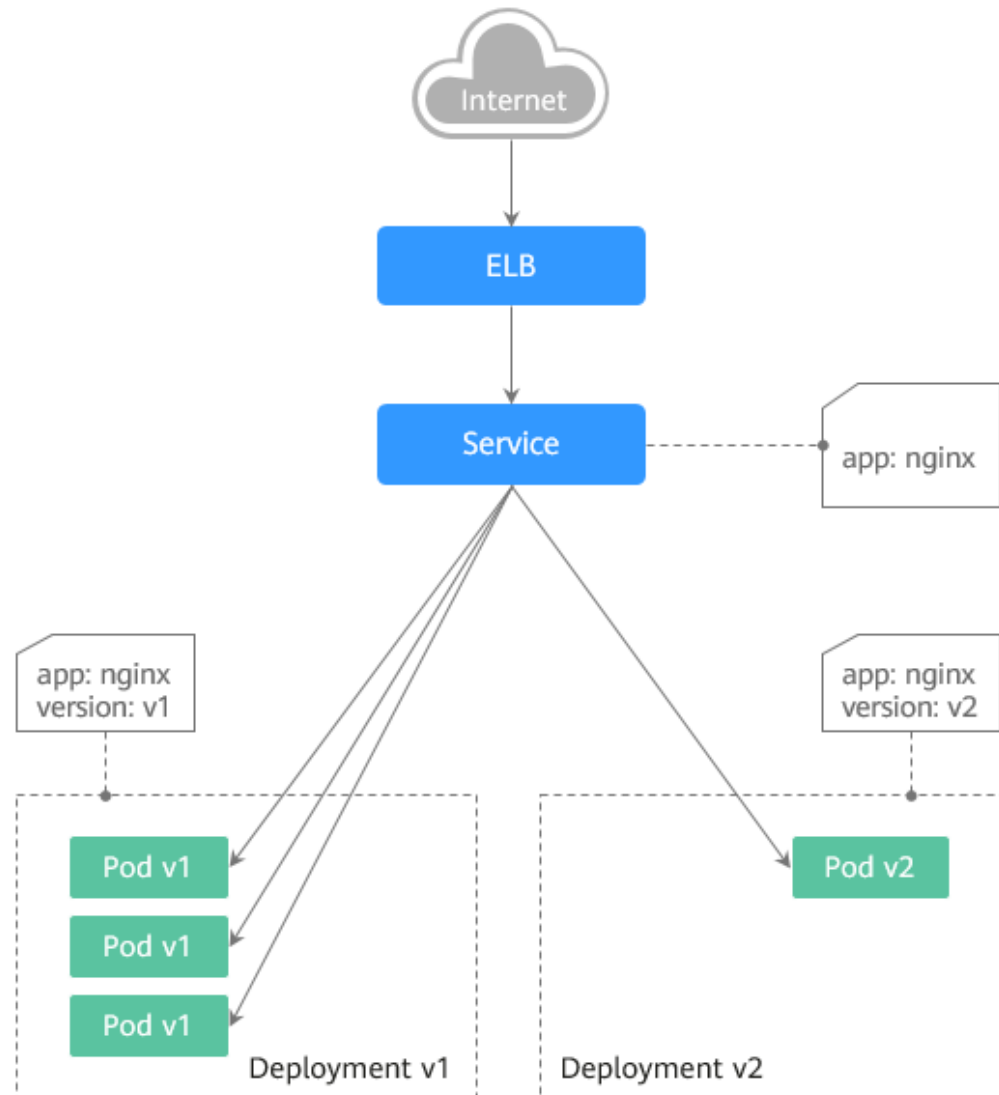
Users usually use Kubernetes objects such as Deployments and StatefulSets to deploy services. Each workload manages a group of pods. The following figure uses Deployment as an example.



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.

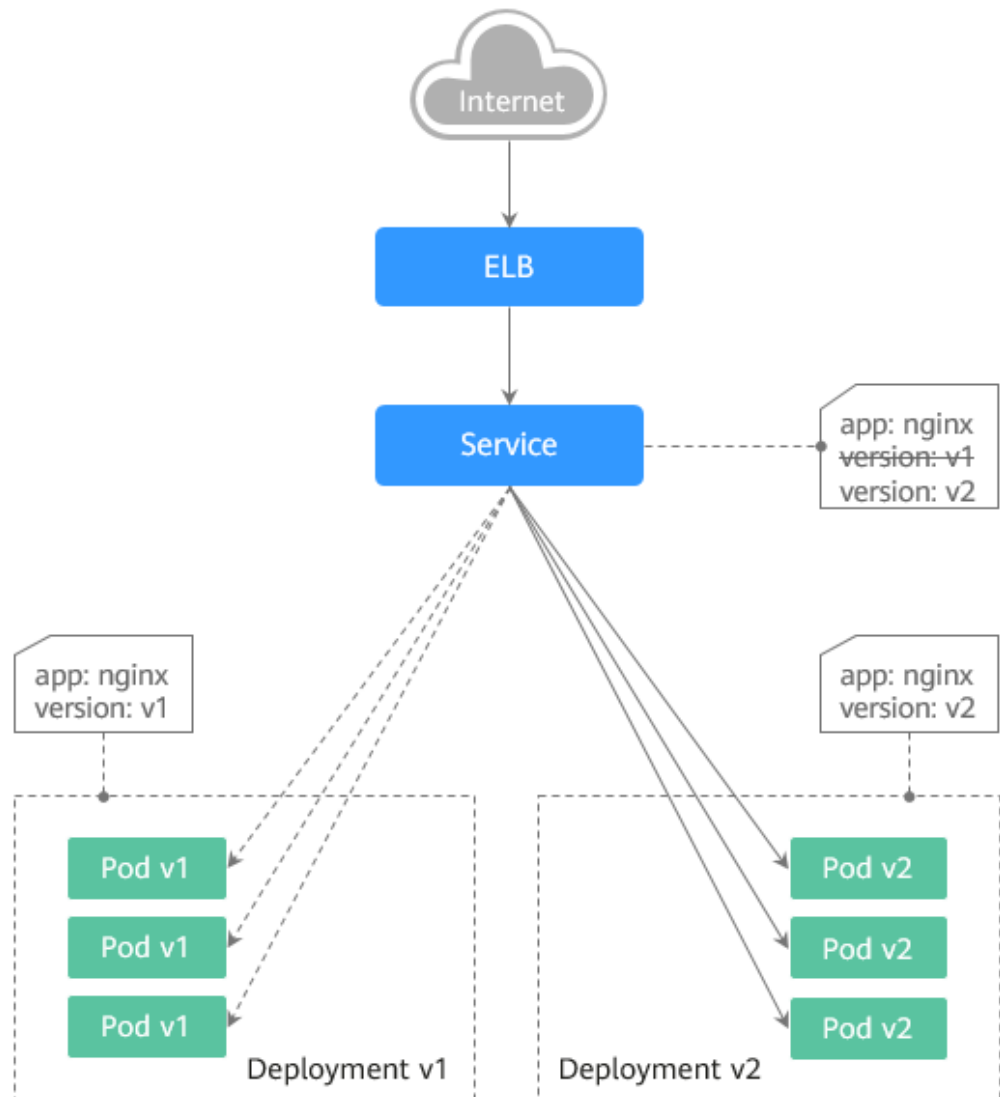
- **Grayscale release principles**

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 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

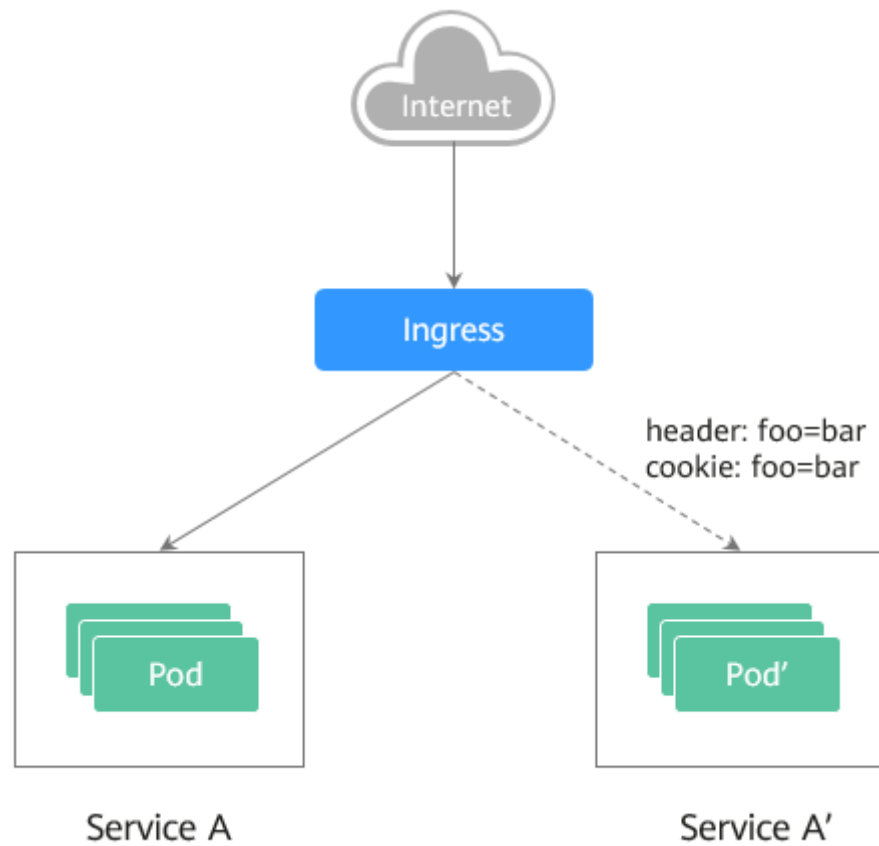
This section describes the scenarios and practices of using Nginx Ingress to implement grayscale release and blue-green deployment.

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:

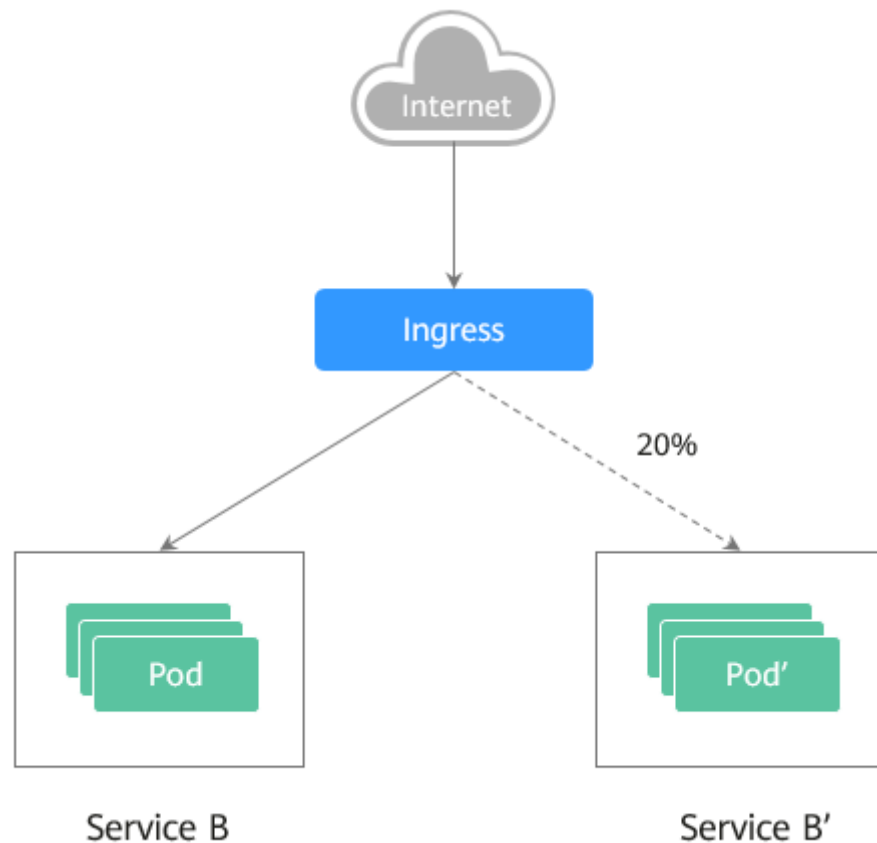
- **Scenario 1: Split some user traffic to the new version.**

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.
- 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/v1
kind: Ingress
metadata:
  name: gray-release
  namespace: default
  annotations:
    kubernetes.io/elb.port: '80'
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - path: /
        backend:
          service:
            name: old-nginx # Set the back-end service to old-nginx.
            port:
              number: 80
          property:
            ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
        ingressClassName: nginx # Nginx ingress is used.
```

- 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/v1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    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:
          service:
            name: new-nginx # Set the back-end service to new-nginx.
            port:
              number: 80
          property:
            ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
        ingressClassName: nginx # Nginx ingress is used.
```

- 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/v1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    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:
              service:
                name: new-nginx # Set the back-end service to new-nginx.
                port:
                  number: 80
            property:
              ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
              pathType: ImplementationSpecific
            ingressClassName: nginx # Nginx ingress is used.
```

- 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/v1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
  annotations:
    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:
          service:
            name: new-nginx # Set the back-end service to new-nginx.
          port:
            number: 80
        property:
          ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
        pathType: ImplementationSpecific
    ingressClassName: nginx # Nginx ingress is used.
```

- 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
```

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/v1
kind: Ingress
metadata:
  name: canary-ingress
  namespace: default
annotations:
```

```
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:
          service:
            name: new-nginx      # Set the back-end service to new-nginx.
            port:
              number: 80
          property:
            ingress.beta.kubernetes.io/url-match-mode: STARTS_WITH
            pathType: ImplementationSpecific
        ingressClassName: nginx # Nginx ingress is used.
```

- 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.