Cloud Container Instance

Best Practice

Issue	01
Date	2024-05-24

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions

Notice

Contents

1 Auto Scaling

1.1 Elastic Scaling of CCE Pods to CCI

CCE Cloud Bursting Engine for CCI functions as a virtual kubelet to connect Kubernetes clusters to APIs of other platforms. This add-on is mainly used to extend Kubernetes APIs to serverless container services such as Huawei Cloud CCI.

With this add-on, you can scale Deployments, StatefulSets, Jobs, and CronJobs running in CCE clusters to **Cloud Container Instance (CCI)** during peak hours. In this way, you can reduce consumption caused by cluster scaling.

Installing the Add-on

1. Log in to the CCE console.
2. Click the name of the target CCE cluster to go to the cluster console.
3. In the navigation pane on the left, choose **Add-ons**.
4. Select the **CCE Cloud Bursting Engine for CCI** add-on and click **Install**.
5. Configure the add-on parameters.

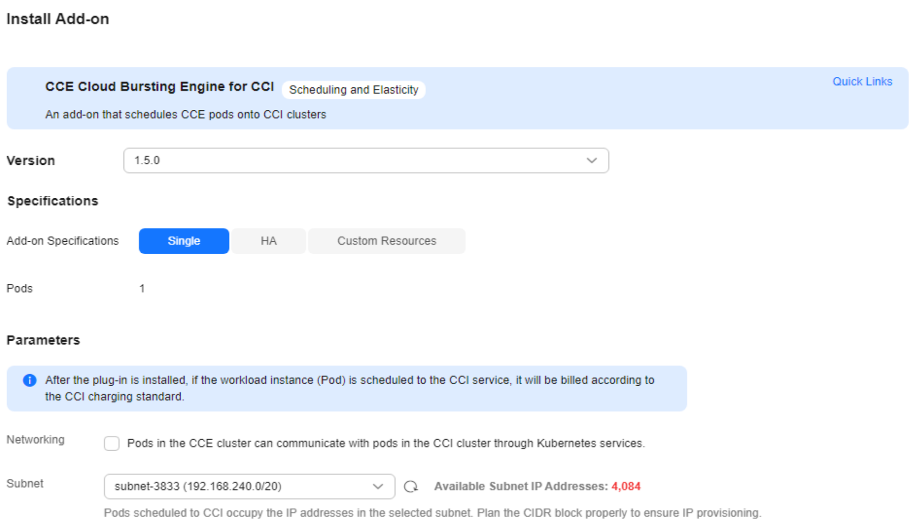
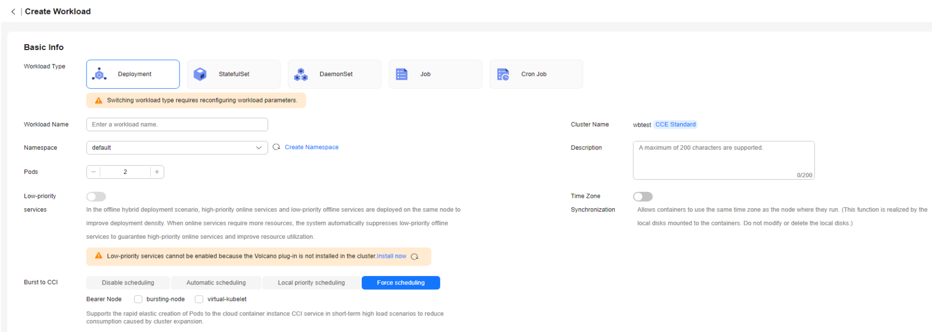


Table 1-1 Add-on parameters

Parameter	Description
Version	Add-on version. There is a mapping between add-on versions and CCE cluster versions. For more details, see "Change History" in **CCE Cloud Bursting Engine for CCI**.
Specifications	Number of pods required for a workload.
Networking	If this option is enabled, pods in a CCE cluster can communicate with the pods in CCI. For details, see **Networking**.

Creating a Workload

1. Log in to the CCE console.
2. Click the name of the target CCE cluster to go to the cluster console.
3. In the navigation pane on the left, choose **Workloads**.
4. Click **Create Workload**. For details, see **Creating a Workload**.
5. Specify basic information. Set **Burst to CCI** to **Force scheduling**. For more information about scheduling policies, see **Scaling Pods to CCI**.



6. Configure the container parameters.
7. Click **Create Workload**.
8. On the **Workloads** page, click the name of the created workload to go to the workload details page.
9. View the node where the workload is running. If the workload is running on a CCI node, it has been scheduled to CCI.

Uninstalling the Add-on

1. Log in to the CCE console.
2. Click the name of the target CCE cluster to go to the cluster console.
3. In the navigation pane on the left, choose **Add-ons**.
4. Select the **CCE Cloud Bursting Engine for CCI** add-on and click **Uninstall**.

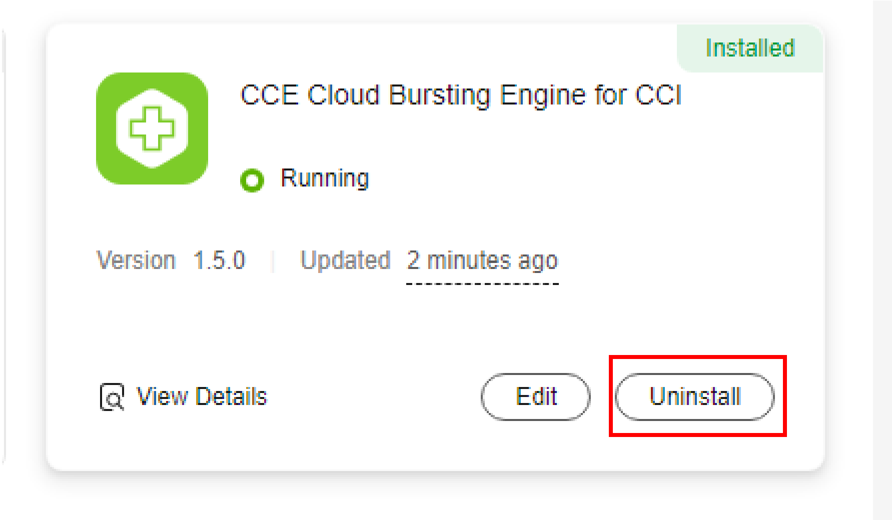


Table 1-2 Special scenarios for uninstalling the add-on

Scenario	Symptom	Description
There are no nodes in the CCE cluster that the bursting add-on needs to be uninstalled from.	Failed to uninstall the bursting add-on.	If the bursting add-on is uninstalled from the cluster, a job for clearing resources will be started in the cluster. To ensure that the job can be started, there is at least one node in the cluster that can be scheduled.
The CCE cluster is deleted, but the bursting add-on is not uninstalled.	There are residual resources in the namespace on CCI. If the resources are not free, additional expenditures will be generated.	The cluster is deleted, but the resource clearing job is not executed. You can manually clear the namespace and residual resources.

For more information about the bursting add-on, see **CCE Cloud Bursting Engine for CCI**.

2 Workload Creation

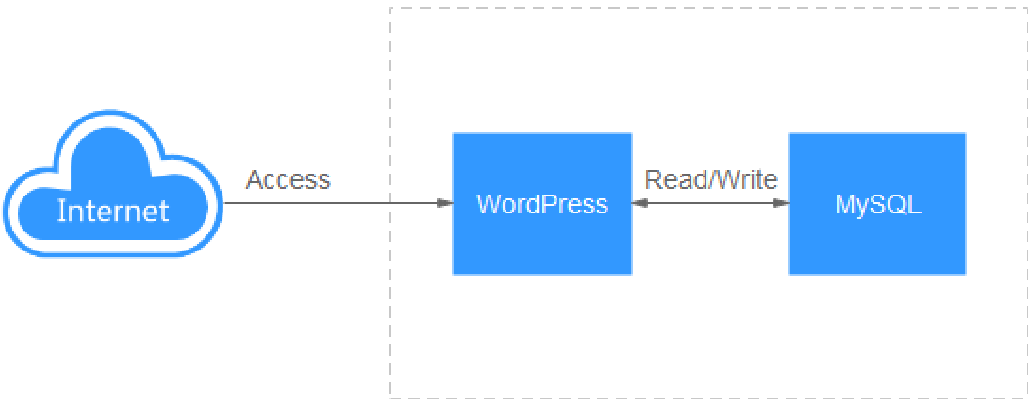
2.1 Overview

You can create workloads on the CCI console or by calling APIs. What are the differences between the two methods and running the **docker run** commands?

This chapter uses the WordPress and MySQL as examples to compare the three methods.

The WordPress is a blog platform developed in hypertext preprocessor (PHP). You can set up your websites on the services that support PHP and MySQL databases, or use the WordPress as a content management system. For more information about the WordPress, visit **https://wordpress.org/**.

The WordPress must be used together with MySQL. The WordPress runs the content management program while MySQL serves as a database to store data. Generally, the WordPress and MySQL run in different containers, as shown in the following figure.



2.2 Using docker run Commands to Run Containers

Docker is an open source engine that manages images and containers. A Docker image includes all dependencies required for running an application. The processes contained in the image are isolated from each other.

Docker containers are built on Docker images.

Preparing Images

WordPress and MySQL images are general-purpose images and can be obtained from the container registry.

You can run the **docker pull** command on the device where the container engine is installed to download images.

```
docker pull mysql:5.7
docker pull wordpress
```

Run the **docker images** command to view the images. As shown in the following figure, two images exist on the local host.

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
wordpress           latest              6a837ea4bd22        6 days ago          408MB
mysql               5.7                 0d16d0a97dd1        5 weeks ago         372MB
```

Running Containers

You can use the container engine to run the WordPress and MySQL containers, and use the **--link** parameter to connect the two containers. In this way, the WordPress container can access the MySQL container without code changes.

Run the following command to run the MySQL container:

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=******** -e MYSQL_DATABASE=wordpress -d
mysql:5.7
```

The parameters are described as follows:

- **--name**: specifies the container name as **some-mysql**.
- **-e**: specifies the environment variable of the container. In this example, the value of **MYSQL_ROOT_PASSWORD** is ************ (replace ******** with your password). The environment variable **MYSQL_DATABASE** indicates the name of the database to be created when the image is started. Its value is **wordpress** in this example.
- **-d**: indicates that the container runs in the backend.

Run the following command to run the WordPress container:

```
docker run --name some-wordpress --link some-mysql:mysql -p 8080:80 -e
WORDPRESS_DB_PASSWORD=******** -e WORDPRESS_DB_USER=root -d wordpress
```

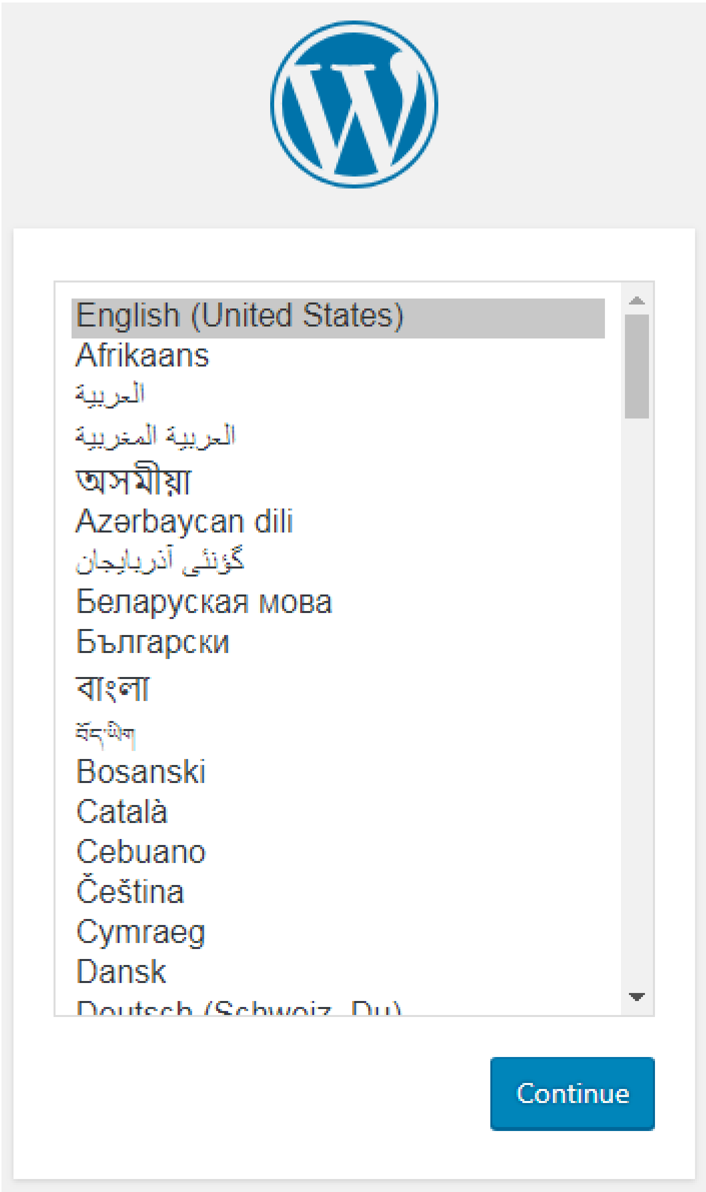
The parameters are described as follows:

- **--name**: specifies the container name as **some-wordpress**.
- **--link**: connects the **some-wordpress** container to the **some-mysql** container and changes the name of the **some-mysql** container to **mysql**. **--link** provides an easy way to connect two containers. Alternatively, you can configure the environment variable **WORDPRESS_DB_HOST** of the **some-wordpress** container to access the IP address and port of the **mysql** container.
- **-p**: specifies ports for mapping. In this example, port 80 of the container is mapped to port 8080 of the host.
- **-e**: specifies the environment variable of the container. In this example, the value of **WORDPRESS_DB_PASSWORD** is ************ (replace ************ with your

password). The value of **WORDPRESS_DB_PASSWORD** must be the same as that of **MYSQL_ROOT_PASSWORD** because the WordPress requires a password to access the MySQL database. **WORDPRESS_DB_USER** indicates the username for accessing the MySQL database. Set it to **root**.

- **-d**: indicates that the container runs in the backend.

After the WordPress runs, you can access WordPress blogs through http://127.0.0.1:8080.



2.3 Using the CCI Console to Create Workloads

Using docker run Commands to Run Containers describes how you can run the WordPress workload by running the **docker run** commands. However, it is not convenient to use a container engine in many scenarios, such as auto scaling and rolling upgrade.

CCI provides a serverless container engine that frees you from managing clusters or servers. CCI delivers agility and high performance with only three steps. It enables you to create stateless workloads (Deployments) and stateful workloads (StatefulSets). It also enhances container security isolation and supports fast workload deployment, elastic load balancing, auto scaling, and blue-green deployment based on the Kubernetes workload model.

Creating a Namespace

Step 1 Log in to the CCI console. In the navigation pane, choose **Namespaces**.

Step 2 Click **Create** for the target namespace type.

Step 3 Enter a namespace name.

Step 4 Configure a VPC.

Select an existing VPC or create one. Recommended CIDR blocks for the new VPC are 10.0.0.0/8-24, 172.16.0.0/12-24, and 192.168.0.0/16-24.

Step 5 Configure a subnet.

Ensure that there are sufficient available IP addresses in the subnet. If IP addresses are insufficient, workload creation will fail.

Step 6 Click **Create**.

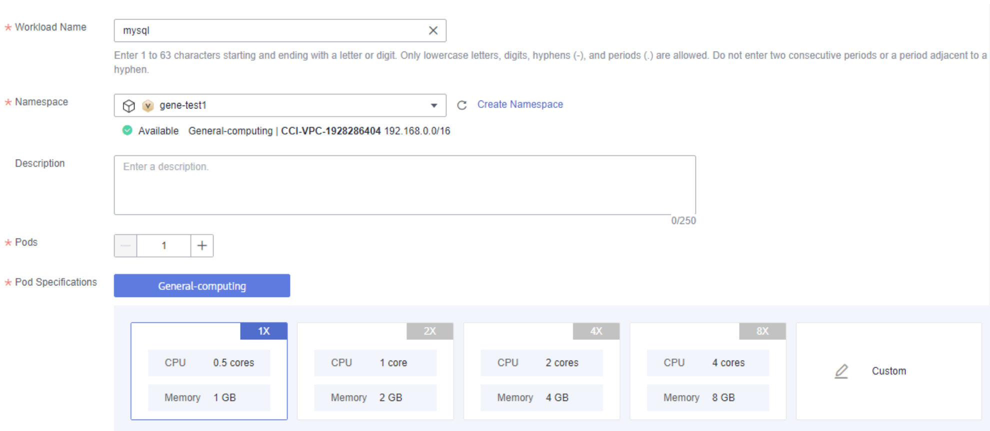
----End

Creating a MySQL Workload

Step 1 Log in to the CCI console. In the navigation pane, choose **Workloads** > **Deployments**. On the page displayed, click **Create from Image**.

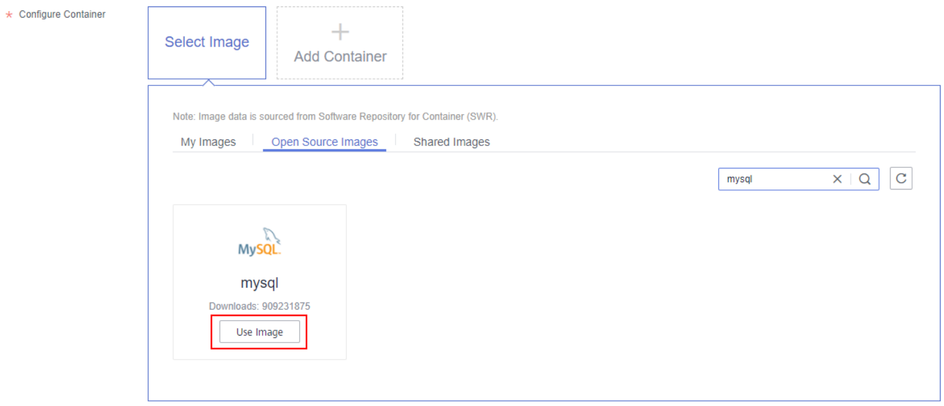
Step 2 Specify basic information.

- **Workload Name**: **mysql**
- **Namespace**: Select the namespace created in **Creating a Namespace**.
- **Pods**: Change the value to **1** in this example.
- **Pod Specifications**: Select the general-computing pod with 0.5-core CPU and 1 GiB of memory.

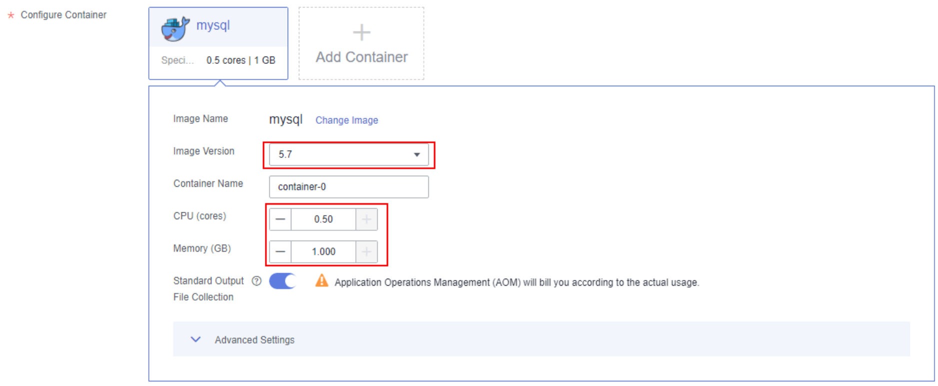


- **Container Settings**

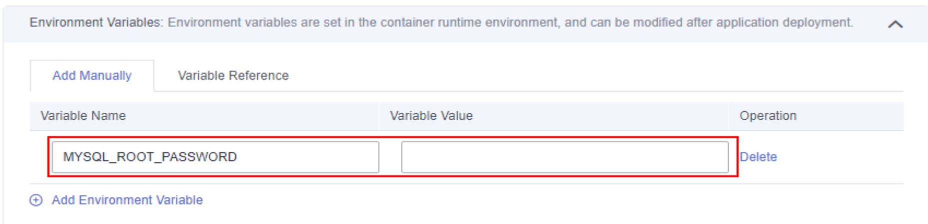
  a. On the **Open Source Images** tab, search for the mysql image and click **Use Image**.



  b. Set image parameters. Specifically, set the image version to **5.7**, CPU to **0.50**, and memory to **1.000**.

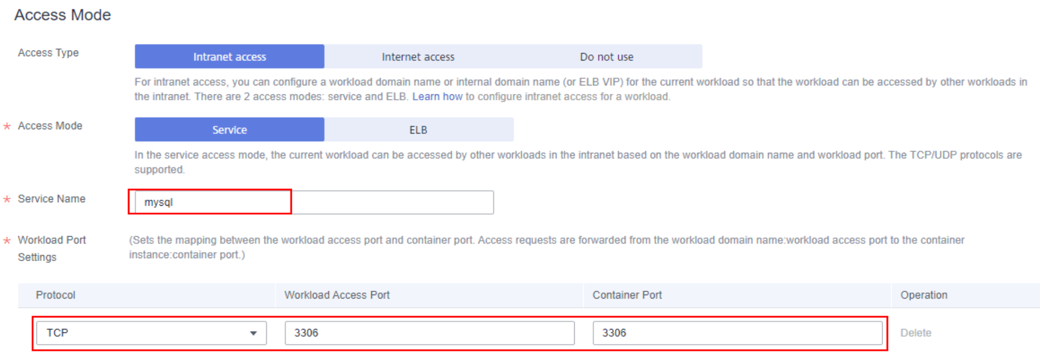


  c. In the advanced settings, enter the environment variable **MYSQL_ROOT_PASSWORD** and its value. The value is the password of the MySQL database. You need to set the password by yourself.



Step 3 Click **Next: Configure Access Settings**. Set **Access Type** to **Intranet access**. In this case, the workload can be accessed by other workloads in CCI by using **Service name:Port**. In addition, set **Service Name** to **mysql**, and map workload access port 3306 to container port 3306 (default access port of the MySQL image).

In this way, other workloads in CCI can access the MySQL workload by using **mysql:3306**.



Step 4 Click **Next**. On the page that is displayed, check the configurations and then click **Submit**.

In the workload list, if the workload is in the **Running** state, the workload is successfully created.

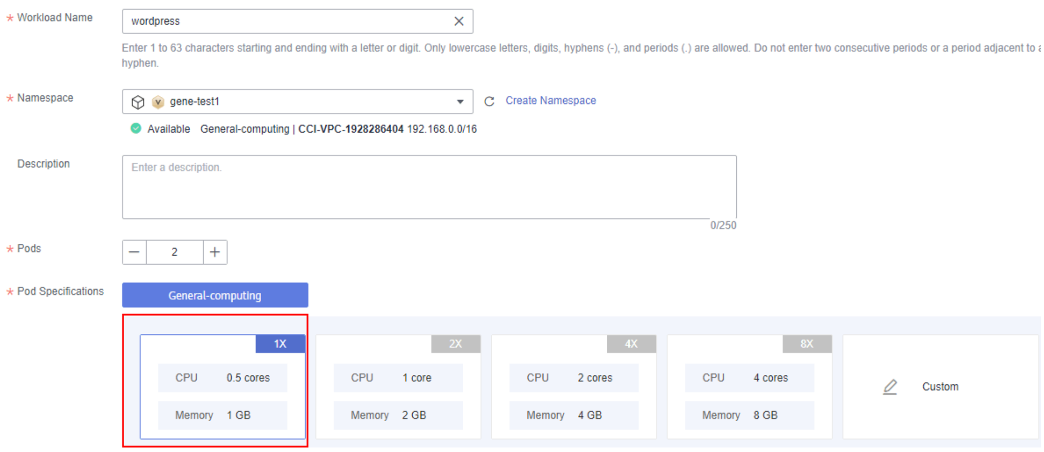
----End

Creating a WordPress Workload

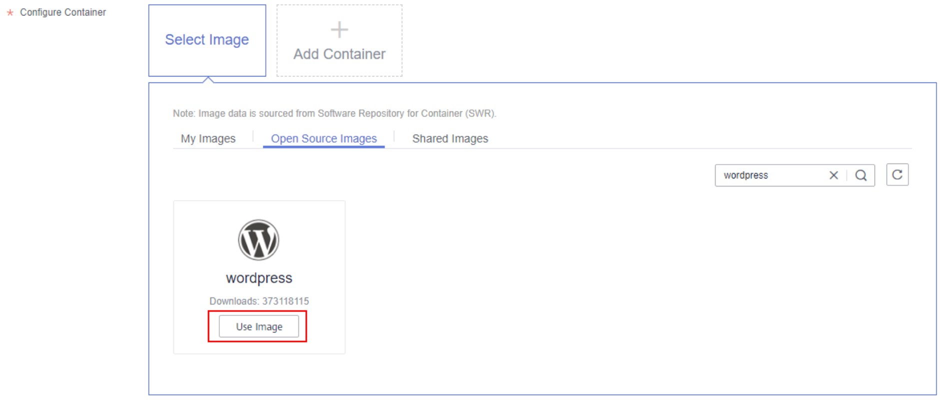
Step 1 Log in to the CCI console. In the navigation pane, choose **Workloads** > **Deployments**. On the page displayed, click **Create from Image**.

Step 2 Specify basic information.

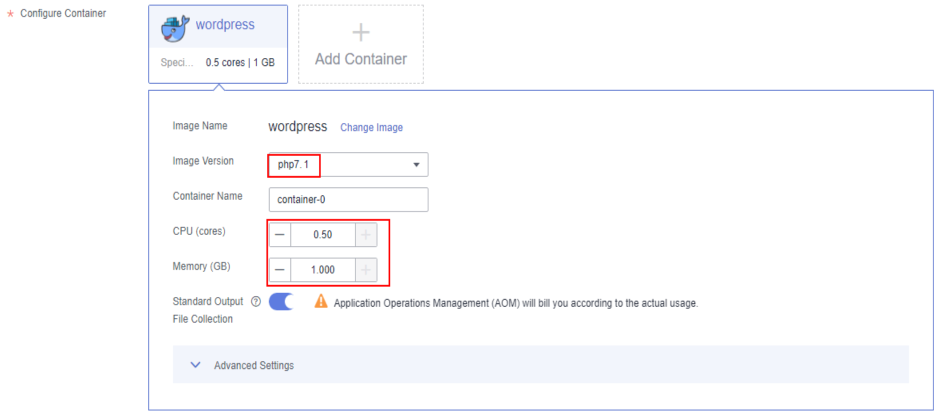
- **Workload Name**: **wordpress**
- **Namespace**: Select the namespace created in **Creating a Namespace**.
- **Pods**: Change the value to **2** in this example.
- **Pod Specifications**: Select the general-computing pod with 0.5-core CPU and 1 GiB of memory.



- **Configure Container**
  a. On the **Open Source Images** tab page, search for the wordpress image and click **Use Image**.



b. Set image parameters. Specifically, set the image version to **php7.1**, CPU to **0.50**, and memory to **1.000**.



c. In the **Advanced Settings** area, expand **Environment Variables** and add environment variables to enable the wordpress application to access the MySQL database.

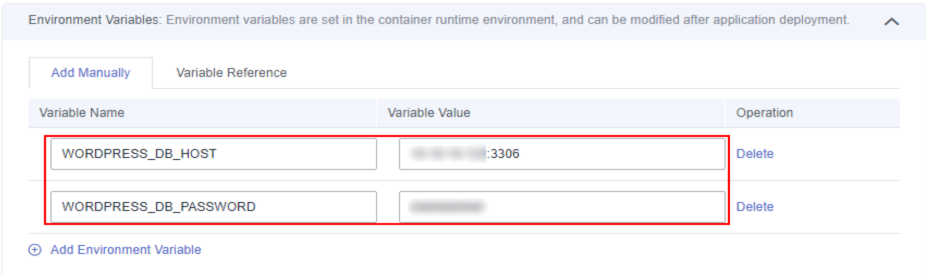
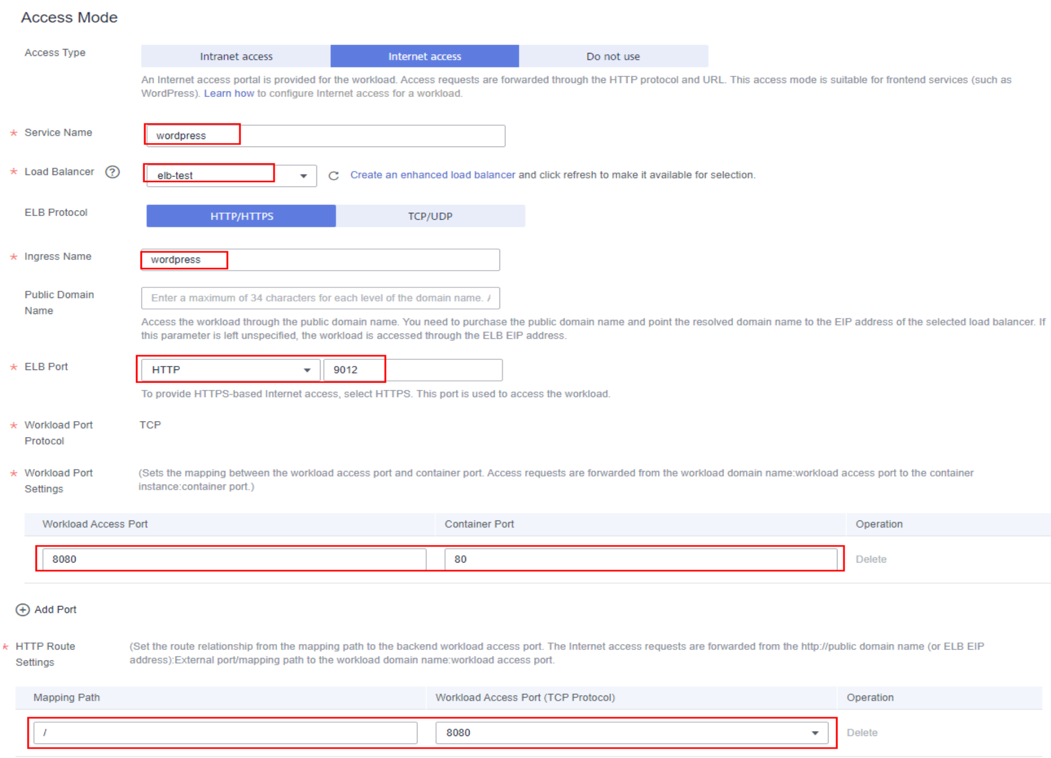


Table 2-1 Description of environment variables

Variable Name	Variable Value/Variable Reference
WORDPRESS_DB_HOST	Address for accessing the MySQL database. Example: 10.***.***.***:3306
WORDPRESS_DB_PASSWORD	Password for accessing the MySQL database. The password must be the same as the MySQL password set in **Creating a MySQL Workload**.

Step 3 Click **Next: Configure Access Settings**.

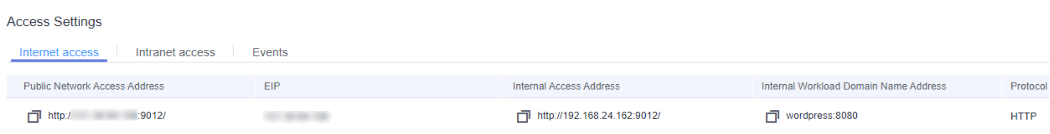
Set **Access Type** to **Internet access** and **Service Name** to **wordpress**, and select a load balancer. If no load balancers are available, click **Create Share Load Balancer**. Set **ELB Protocol** to **HTTP** and **ELB Port** to **9012**. In the **Workload Port Settings** area, set a mapping between workload access port 8080 and container port 80 (default access port in the WordPress image). In the **HTTP Route Settings** area, set **Mapping Path** to **/** (so **http://***Load balancer IP address*:*Port* can be used to access the WordPress) and **Workload Access Port** to **8080**.



Step 4 Click **Next**. On the page that is displayed, check the configurations and then click **Submit**.

In the workload list, if the workload is in the **Running** state, the workload is successfully created. In this case, you can click the workload to go to its details page.

In the **Access Settings** area, click **Internet Access** and view the access address (*Load balancer IP address*:*Port*).



----End

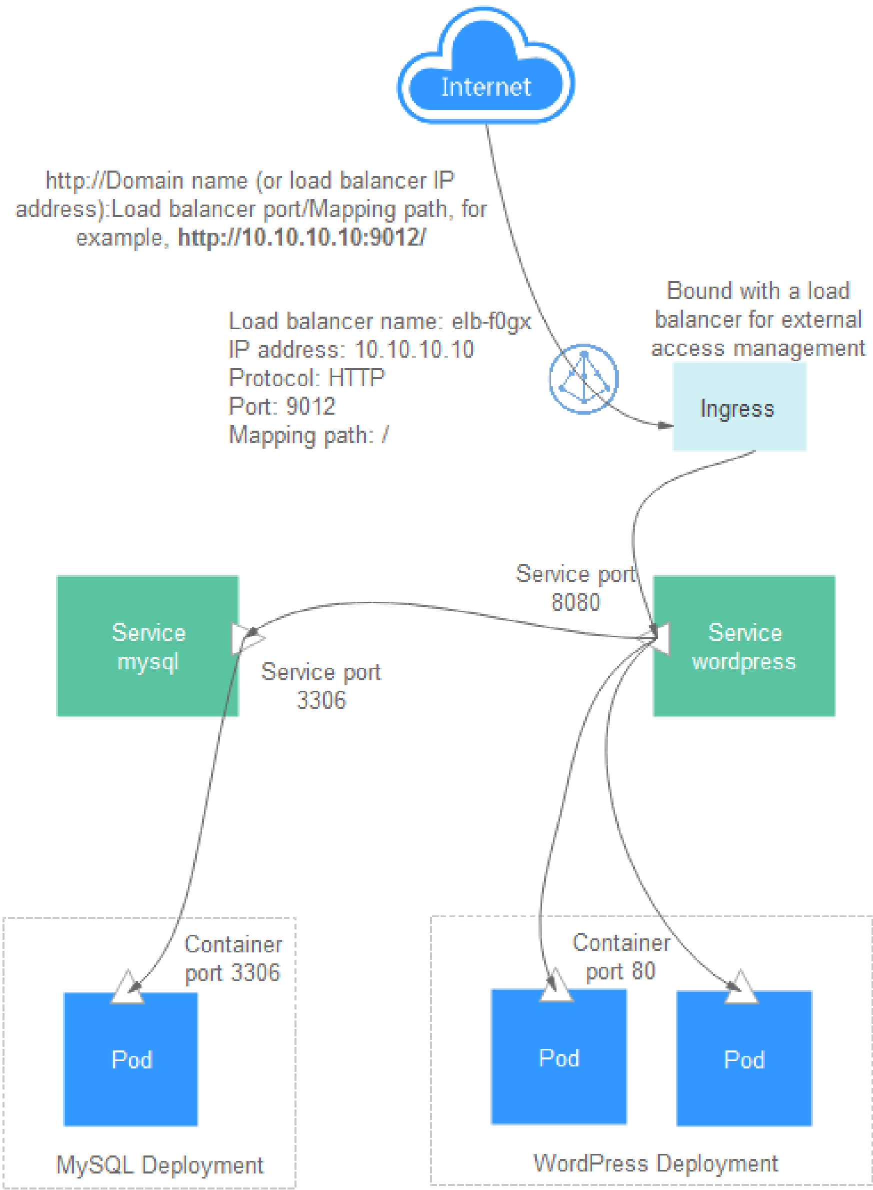
2.4 Calling APIs to Create Workloads

CCI supports Kubernetes APIs. Compared with using the console to create workloads, calling APIs is much easier.

In Kubernetes, a pod is the minimum unit for container running and can encapsulate one or more containers, storage resources, and an independent network IP address. In practice, pods are rarely created directly. Kubernetes uses controllers such as Deployment and StatefulSet to manage pods. In addition, Kubernetes uses Services to define pods and their access policies, and uses ingresses to manage external access. For more information about Kubernetes resources, see ***Cloud Container Instance Developer Guide***.

For the WordPress application, you can call APIs to create a series of resources, as shown in the following figure.

- MySQL: Create a Deployment to deploy the MySQL, and create a Service to define the access policy of the MySQL.
- WordPress: Create a Deployment to deploy the WordPress, and create a Service and an ingress to define the access policy of the WordPress.



Namespace

Step 1 Call the API described in **Creating a Namespace** to create a namespace and specify a namespace type.

```
{
    "apiVersion": "v1",
    "kind": "Namespace",
    "metadata": {
        "name": "namespace-test",
        "annotations": {
            "namespace.kubernetes.io/flavor": "gpu-accelerated"
        }
    },
    "spec": {
        "finalizers": [
            "kubernetes"
        ]
    }
}
```

Step 2 Call the API described in **Creating a Network** to create a network, and associate the network with a Virtual Private Cloud (VPC) and subnet.

```
{
    "apiVersion": "networking.cci.io/v1beta1",
    "kind": "Network",
    "metadata": {
        "annotations": {
            "network.alpha.kubernetes.io/default-security-group": "{{security-group-id}}",
            "network.alpha.kubernetes.io/domain-id": "{{domain-id}}",
            "network.alpha.kubernetes.io/project-id": "{{project-id}}"
        },
        "name": "test-network"
    },
    "spec": {
        "availableZone": "{{zone}}",
        "cidr": "192.168.0.0/24",
        "attachedVPC": "{{vpc-id}}",
        "networkID": "{{network-id}}",
        "networkType": "underlay_neutron",
        "subnetID": "{{subnet-id}}"
    }
}
```

----End

MySQL

Step 1 Call the API described in **Creating a Deployment** to deploy the MySQL.

- Set the Deployment name to **mysql**.
- Set the pod label to **app:mysql**.
- Use the **mysql:5.7** image.
- Set the value of the environment variable **MYSQL_ROOT_PASSWORD** to ************ (replace ************ with your password).

```
{
    "apiVersion": "apps/v1",
    "kind": "Deployment",
    "metadata": {
        "name": "mysql"
    },
    "spec": {
        "replicas": 1,
        "selector": {
```

```
            "matchLabels": {
                "app": "mysql"
            }
        },
        "template": {
            "metadata": {
                "labels": {
                    "app": "mysql"
                }
            },
            "spec": {
                "containers": [
                    {
                        "image": "mysql:5.7",
                        "name": "container-0",
                        "resources": {
                            "limits": {
                                "cpu": "500m",
                                "memory": "1024Mi"
                            },
                            "requests": {
                                "cpu": "500m",
                                "memory": "1024Mi"
                            }
                        },
                        "env": [
                            {
                                "name": "MYSQL_ROOT_PASSWORD",
                                "value": "********"
                            }
                        ]
                    }
                ],
                "imagePullSecrets": [
                    {
                        "name": "imagepull-secret"
                    }
                ]
            }
        }
    }
}
```

Step 2 Call the API described in **Creating a Service** to create a Service, and define the access policy for the pod created in **Step 1**.

- Set the Service name to **mysql**.
- Select the pod whose label is **app:mysql** to associate the pod created in **Step 1**.
- Map workload access port 3306 to container port 3306.
- The access type of the Service is **ClusterIP**, that is, ClusterIP is used to access the Service inside the cluster.

```
{
    "apiVersion": "v1",
    "kind": "Service",
    "metadata": {
        "name": "mysql",
        "labels": {
            "app": "mysql"
        }
    },
    "spec": {
        "selector": {
            "app": "mysql"
        },
        "ports": [
            {
```

```
                "name": "service0",
                "targetPort": 3306,
                "port": 3306,
                "protocol": "TCP"
            }
        ],
        "type": "ClusterIP"
    }
}
```

----End

WordPress

Step 1 Call the API described in **Creating a Deployment** to deploy the WordPress.

- Set the Deployment name to **wordpress**.
- Set the value of replicas to **2**, indicating that two pods are created.
- Set the pod label to **app:wordpress**.
- Use the **wordpress:latest** image.
- Set the value of the environment variable **WORDPRESS_DB_PASSWORD** to ************ (replace ************ with your password). This password must be the same as **MYSQL_ROOT_PASSWORD** set for the MySQL.

```
{
    "apiVersion": "apps/v1",
    "kind": "Deployment",
    "metadata": {
        "name": "wordpress"
    },
    "spec": {
        "replicas": 2,
        "selector": {
            "matchLabels": {
                "app": "wordpress"
            }
        },
        "template": {
            "metadata": {
                "labels": {
                    "app": "wordpress"
                }
            },
            "spec": {
                "containers": [
                    {
                        "image": "wordpress:latest",
                        "name": "container-0",
                        "resources": {
                            "limits": {
                                "cpu": "500m",
                                "memory": "1024Mi"
                            },
                            "requests": {
                                "cpu": "500m",
                                "memory": "1024Mi"
                            }
                        },
                        "env": [
                            {
                                "name": "WORDPRESS_DB_PASSWORD",
                                "value": "********"
                            }
                        ]
                    }
                ],
```

```
                "imagePullSecrets": [
                    {
                        "name": "imagepull-secret"
                    }
                ]
            }
        }
    }
}
```

Step 2 Call the API described in **Creating a Service** to create a Service, and define the access policy for the pod created in **Step 1**.

- Set the Service name to **wordpress**.
- Select the pod whose label is **app:wordpress** to associate the pod created in **Step 1**.
- Map workload access port 8080 to container port 80. For the WordPress image, port 80 is the default externally exposed port.
- The access type of the Service is **ClusterIP**, that is, ClusterIP is used to access the Service inside the cluster.

```
{
    "apiVersion": "v1",
    "kind": "Service",
    "metadata": {
        "name": "wordpress",
        "labels": {
            "app": "wordpress"
        }
    },
    "spec": {
        "selector": {
            "app": "wordpress"
        },
        "ports": [
            {
                "name": "service0",
                "targetPort": 80,
                "port": 8080,
                "protocol": "TCP"
            }
        ],
        "type": "ClusterIP"
    }
}
```

Step 3 Call the API described in **Creating an Ingress** to create an ingress to define the external access policy of the WordPress. In this step, you need to configure a load balancer that is in the same VPC as the WordPress.

- metadata.annotations.kubernetes.io/elb.id: ID of the load balancer
- metadata.annotations.kubernetes.io/elb.ip: IP address of the load balancer
- metadata.annotations.kubernetes.io/elb.port: Port configured for the load balancer
- spec.rules: a set of rules for accessing the Service. **path** lists the paths for accessing the Service, for example, "/". Each path is associated with a backend (for example, **wordpress:8080**). A backend represents a combination of service:port. Ingress traffic will be forwarded to the corresponding backend.

After the configuration is complete, the traffic destined for the load balancer (*Load balancer IP address:Port*) is transmitted to the wordpress:8080 Service. Because the Service is associated with the WordPress pod, the traffic finally accesses the WordPress container deployed in **Step 1**.

```
{
    "apiVersion": "extensions/v1beta1",
    "kind": "Ingress",
    "metadata": {
        "name": "wordpress",
        "labels": {
            "app": "wordpress",
            "isExternal": "true",
            "zone": "data"
        },
        "annotations": {
            "kubernetes.io/elb.id": "2d48d034-6046-48db-8bb2-53c67e8148b5",
            "kubernetes.io/elb.ip": "10.10.10.10",
            "kubernetes.io/elb.port": "9012"
        }
    },
    "spec": {
        "rules": [
            {
                "http": {
                    "paths": [
                        {
                            "path": "/",
                            "backend": {
                                "serviceName": "wordpress",
                                "servicePort": 8080
                            }
                        }
                    ]
                }
            }
        ]
    }
}
```


----End

2.5 Configuring Dockerfile Parameters for CCI

Scenario

Dockerfiles are generally used to customize images. A Dockerfile is a text file that contains instructions, each of which builds an image layer.

This topic describes the CCI settings corresponding to the Dockerfile configurations.

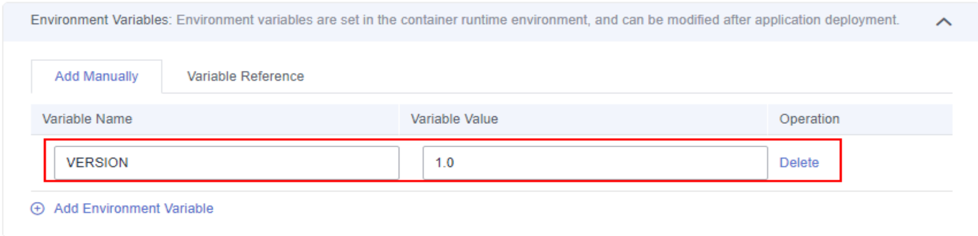
Using Dockerfile Parameters in CCI

The following uses an example to describe the relationship between them.

```
FROM ubuntu:16.04

ENV VERSION 1.0

VOLUME /var/lib/app

EXPOSE 80

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

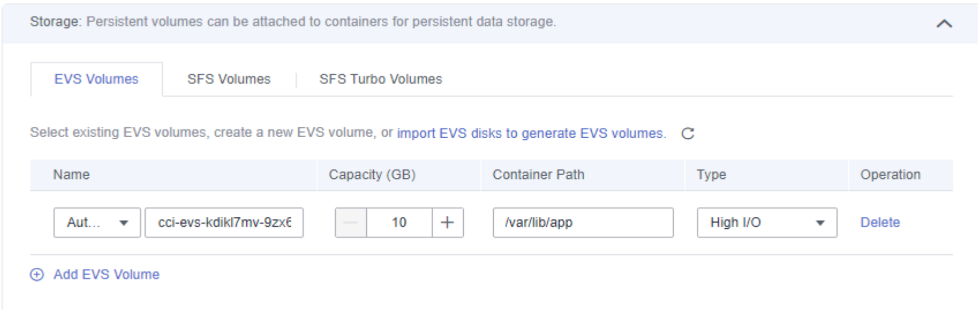
In the preceding example, the Dockerfile contains common parameters, including **ENV**, **VOLUME**, **EXPOSE**, **ENTRYPOINT**, and **CMD**. These parameters can be configured for CCI as follows:

- **ENV** indicates an environment variable. Corresponding to **ENV VERSION 1.0** in the Dockerfile, set **Environment Variables** in the advanced settings as follows when creating a workload on the CCI console.

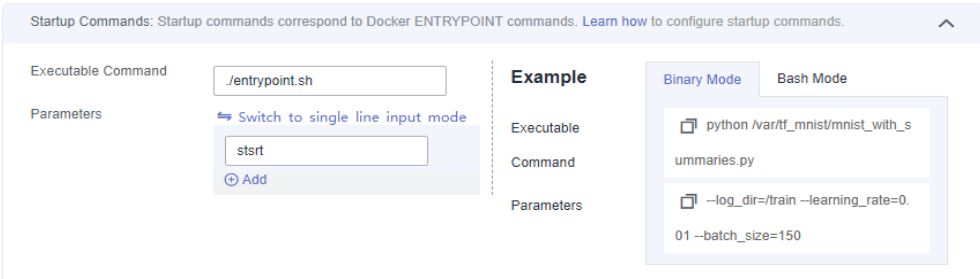


- **VOLUME** indicates a container volume. Generally, this parameter is used together with **docker run -v host path:container volume path**.

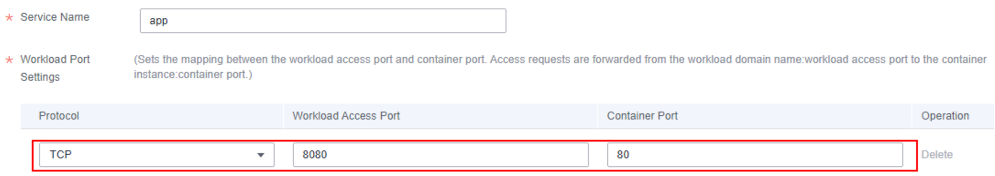
  For CCI, Elastic Volume Service (EVS) disks can be mounted to containers. You only need to add EVS volumes, and configure their sizes and mount paths (that is, container volume paths) when creating workloads.



- **ENTRYPOINT** and **CMD** correspond to the startup command of the advanced settings in CCI. For details, see **Setting Container Startup Commands**.



- **EXPOSE** indicates an exposed port. Generally, this parameter is used together with **docker run -p <host port>:<container port>** when a container is started. To set an exposed port for a CCI container, you only need to configure the **Workload access port:Container port** when creating a workload. In this way, you can access the container through the **Workload domain name:Workload access port**.


* Service Name
app
* Workload Port Settings
(Sets the mapping between the workload access port and container port. Access requests are forwarded from the workload domain name:workload access port to the container instance:container port.)
Protocol
Workload Access Port
Container Port
Operation
TCP
8080
80
Delete

3 Workload Management

3.1 Performing Graceful Rolling Upgrade for CCI Applications

Scenario

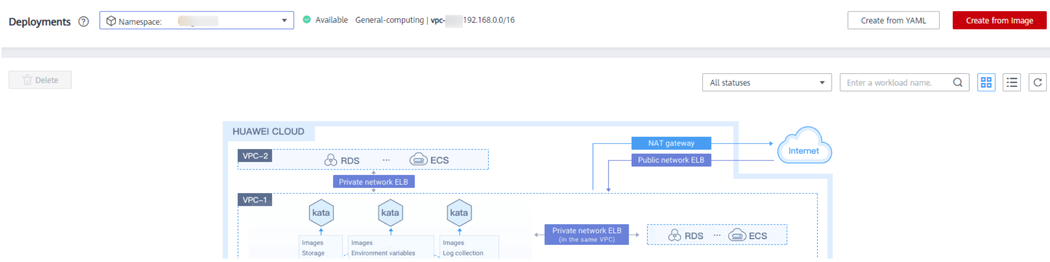
When you deploy a workload in CCI to run an application, the application is exposed as a LoadBalancer Service or ingress, and connected to a dedicated ELB load balancer to allow access traffic to reach the containers directly. When rolling upgrade or auto scaling is performed on the application, your pods may fail to work with ELB and 5xx errors may occur. This section guides you to configure container probes and readiness time to achieve graceful upgrade and auto scaling.

Procedure

The following uses an Nginx Deployment as an example.

Step 1 On the CCI console, choose **Workloads** > **Deployments** in the navigation pane, and click **Create from Image** in the upper right corner.

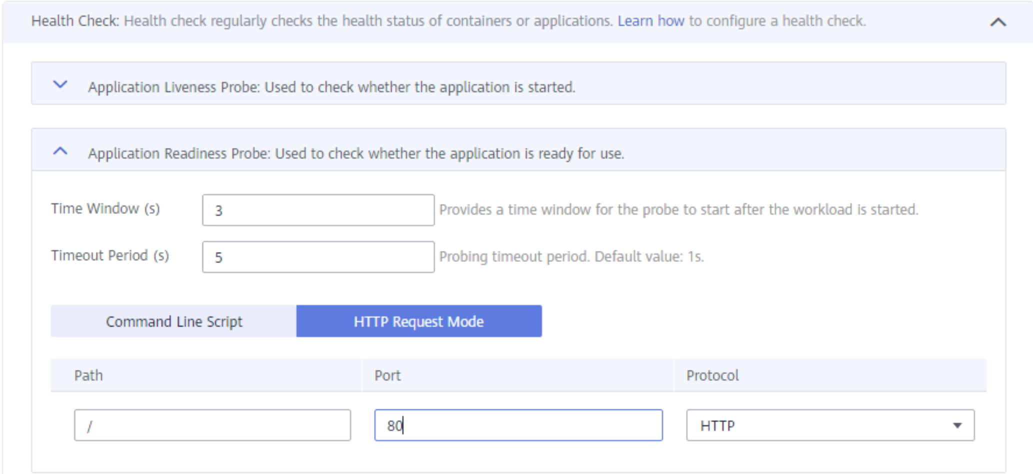
Figure 3-1 Creating a Deployment



Step 2 In the **Container Settings** area, click **Use Image** to select an image.

Step 3 Click **Advanced Settings** of the image, click **Health Check** > **Application Readiness Probe**, and configure the probe.

Figure 3-2 Configuring the application readiness probe

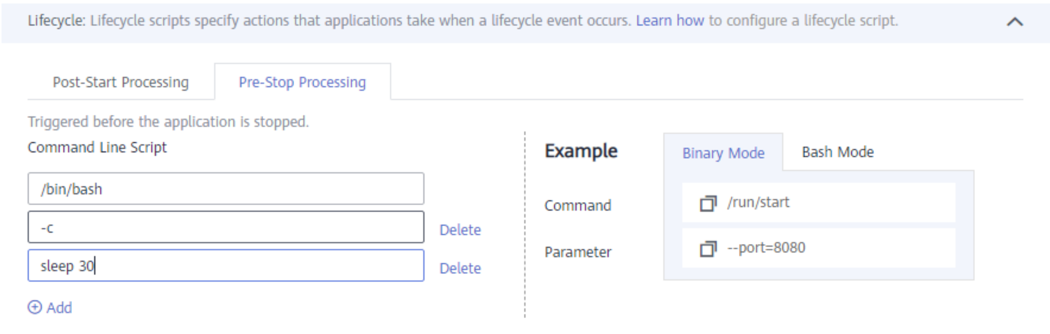


NOTE

The probe checks whether your container is ready. If the container is not ready, requests will not be forwarded to the container.

Step 4 Expand **Lifecycle** and configure the parameters of **Pre-Stop Processing** for the container.

Figure 3-3 Configuring lifecycle parameters

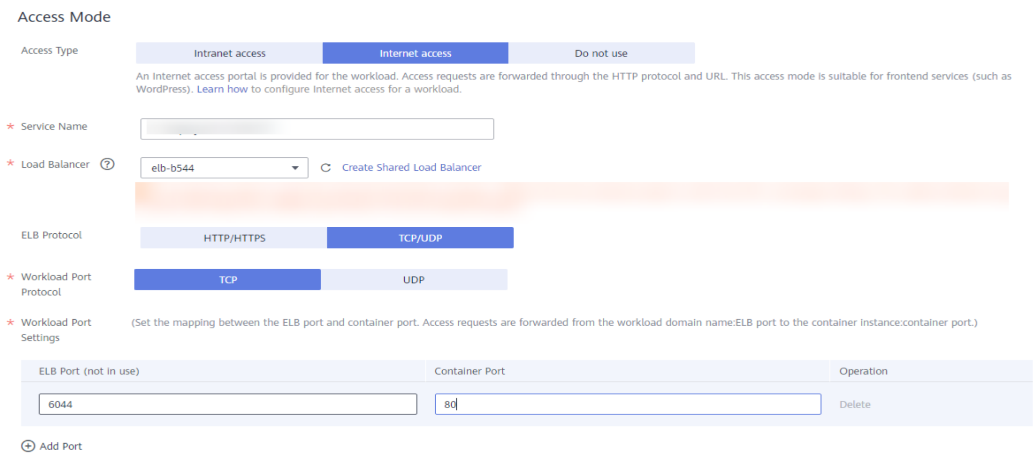


NOTE

This configuration ensures that the container can provide services for external systems during its exit.

Step 5 Click **Next: Configure Access Settings** and configure settings as shown in **Figure 3-4**.

Figure 3-4 Configuring the access type and port



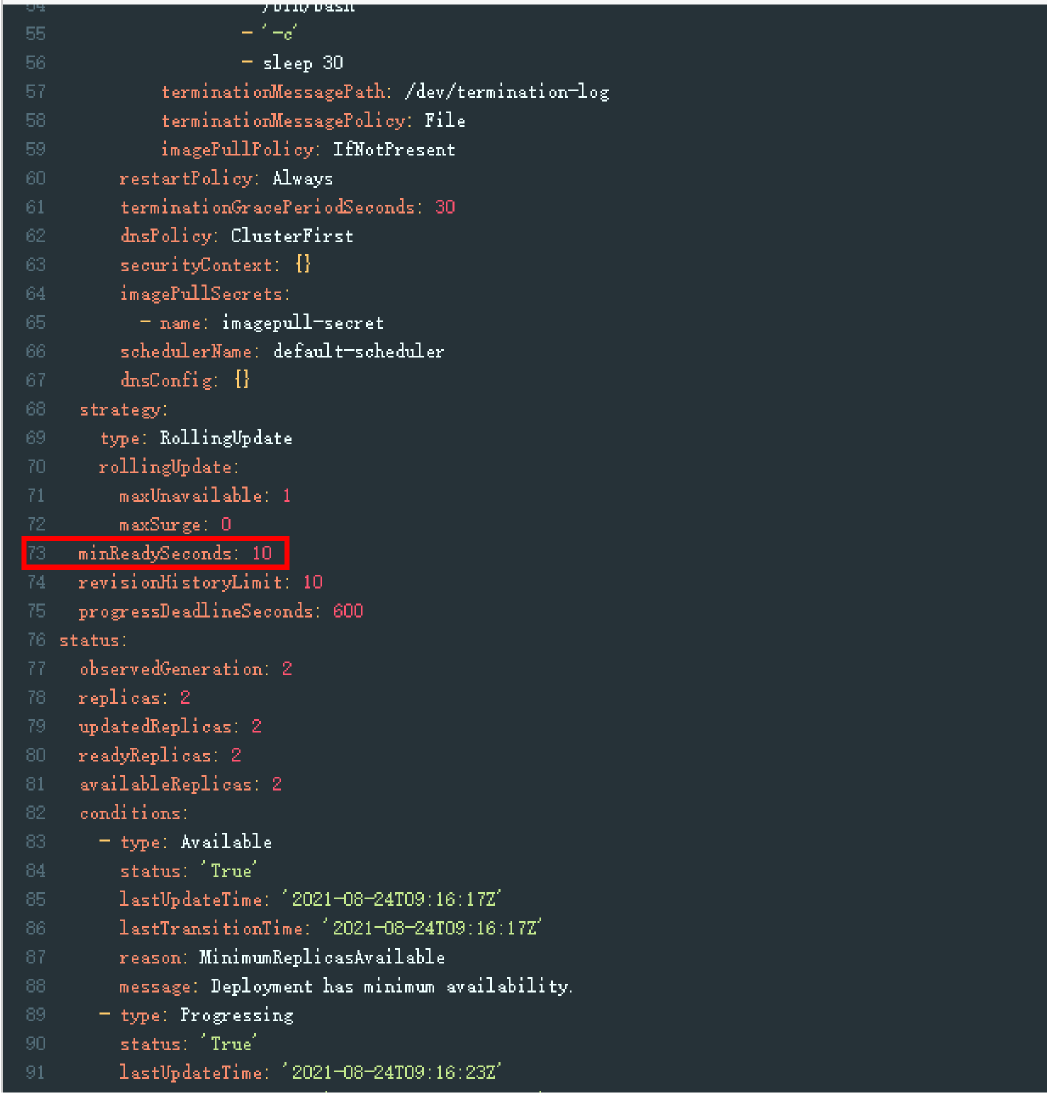
Step 6 Click **Next** and complete the Deployment creation.

Step 7 Configure the minimum readiness time.

A pod is considered available only when the minimum readiness time is exceeded without any of its containers crashing.

In the upper right corner of the **Deployments** page, click **Create YAML** to configure the minimum readiness time as below.

Figure 3-5 Configuring the minimum readiness time



```
                  /bin/bash
                - '-c'
                - sleep 30
            terminationMessagePath: /dev/termination-log
            terminationMessagePolicy: File
            imagePullPolicy: IfNotPresent
        restartPolicy: Always
        terminationGracePeriodSeconds: 30
        dnsPolicy: ClusterFirst
        securityContext: {}
        imagePullSecrets:
          - name: imagepull-secret
        schedulerName: default-scheduler
        dnsConfig: {}
    strategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 1
        maxSurge: 0
    minReadySeconds: 10
    revisionHistoryLimit: 10
    progressDeadlineSeconds: 600
  status:
    observedGeneration: 2
    replicas: 2
    updatedReplicas: 2
    readyReplicas: 2
    availableReplicas: 2
    conditions:
      - type: Available
        status: 'True'
        lastUpdateTime: '2021-08-24T09:16:17Z'
        lastTransitionTime: '2021-08-24T09:16:17Z'
        reason: MinimumReplicasAvailable
        message: Deployment has minimum availability.
      - type: Progressing
        status: 'True'
        lastUpdateTime: '2021-08-24T09:16:23Z'
```

NOTE

- The recommended value of **minReadySeconds** is the expected time for starting the service container plus the duration from the time when the ELB service delivers the member to the time when the member takes effect.
- The value of **minReadySeconds** must be smaller than that of **sleep** to ensure that the new container is ready before the old container stops and exits.

Step 8 Test the application upgrade and auto scaling.

Prepare a client outside the cluster, and configure the detection script **detection_script.sh** with the following content (**100.85.125.90:7552** indicates the public network address for accessing the Service):

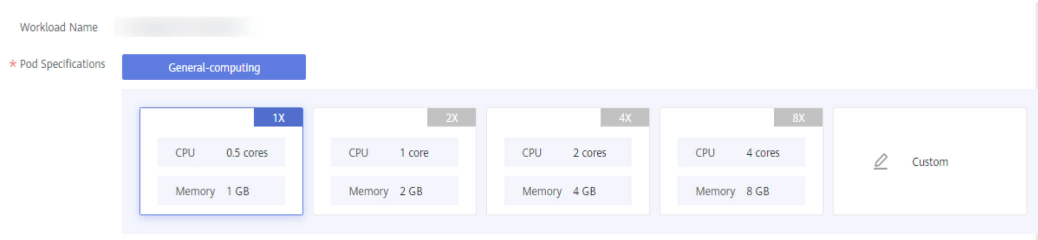
```
#! /bin/bash
for (( ; ; ))
do
        curl -I 100.85.125.90:7552 | grep "200 OK"
        if [ $? -ne 0 ]; then
                echo "response error!"
                exit 1
```



```
        fi
done
```

Step 9 Run the detection script (**bash detection_script.sh**) and trigger the rolling upgrade of the application on the CCI console. You can change the specifications of the container to trigger the rolling upgrade of the application.

Figure 3-6 Modifying container specifications



If the access to the application is not interrupted, and the returned responses are all **200OK**, the graceful upgrade is successfully triggered.

----End

3.2 Exposing Basic Pod Information to Containers Through Environment Variables

If you want a pod to expose its basic information to containers running in the pod, you can use the Kubernetes ***Downward API*** to inject environment variables. This section describes how to add environment variables to the definition of a Deployment or a pod to obtain the namespace, name, UID, IP address, region, and AZ of the pod.

When CCI creates a pod and allocates it to a node, the region and AZ information of the node is added to the pod's annotations.

In this case, the format of the pod's annotations is as follows:

```
apiVersion: v1
 kind: Pod
 metadata:
   annotations:
     topology.kubernetes.io/region: "{{region}}"
     topology.kubernetes.io/zone: "{{available-zone}}"
```

topology.kubernetes.io/region indicates the region of the node.

topology.kubernetes.io/zone indicates the AZ of the node.

Deployment Configuration Example

The following example shows how to use environment variables to obtain basic pod information.

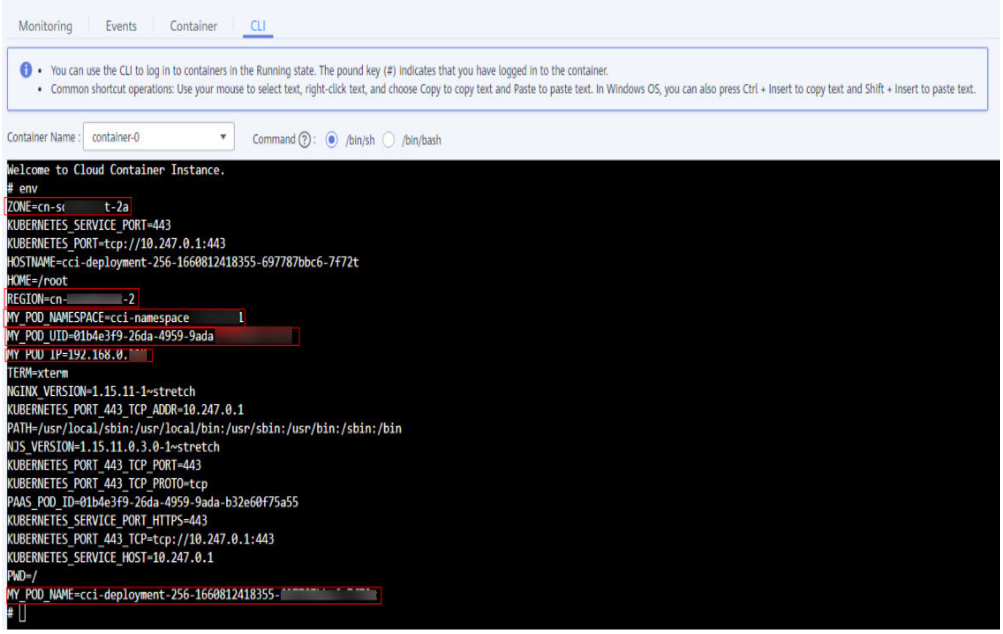
```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: cci-downwardapi-test
  namespace: cci-test # Enter a specific namespace.
```



```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cci-downwardapi-test
  template:
    metadata:
      labels:
        app: cci-downwardapi-test
    spec:
      containers:
        - name: container-0
          image: 'library/euleros:latest'
          command:
            - /bin/bash
            - '-c'
            - while true; do echo hello; sleep 10; done
          env:
           - name: MY_POD_UID
             valueFrom:
               fieldRef:
                 fieldPath: metadata.uid
           - name: MY_POD_NAME
             valueFrom:
               fieldRef:
                 fieldPath: metadata.name
           - name: MY_POD_NAMESPACE
             valueFrom:
               fieldRef:
                 fieldPath: metadata.namespace
           - name: MY_POD_IP
             valueFrom:
               fieldRef:
                 fieldPath: status.podIP
           - name: REGION
             valueFrom:
               fieldRef:
                 fieldPath: metadata.annotations['topology.kubernetes.io/region']
           - name: ZONE
             valueFrom:
               fieldRef:
                 fieldPath: metadata.annotations['topology.kubernetes.io/zone']
          resources:
            limits:
              cpu: 500m
              memory: 1Gi
            requests:
              cpu: 500m
              memory: 1Gi
```

When the Deployment starts, you can view the pod information exposed to the container through environment variables.

Figure 3-7 Basic pod Information



3.3 Configuring Kernel Parameters

CCI uses Kata containers to build an industry-leading serverless container platform. Kata containers are isolated from the physical machine system kernel. They do not affect each other. kernel parameter optimization is a common practice in advanced service deployment scenarios. In a safe situation, CCI allows you to configure kernel parameters through a security context of a pod based on the solution recommended by the Kubernetes community, greatly improving the flexibility of service deployment. For details of security contexts, see ***Configure a Security Context for a Pod or Container***.

In Linux, kernel parameters are usually configured through the sysctl interface. In Kubernetes, kernel parameters are configured through the sysctl security context of the pod. For details of sysctl, see ***Using sysctls in a Kubernetes Cluster***. The security context is applied to all containers in the pod.

CCI allows you to modify the following kernel parameters:

```
kernel.shm*,
kernel.msg*,
kernel.sem,
fs.mqueue.*,
net.* (excluding net.netfilter.* and net.ipv4.vs.*)
```

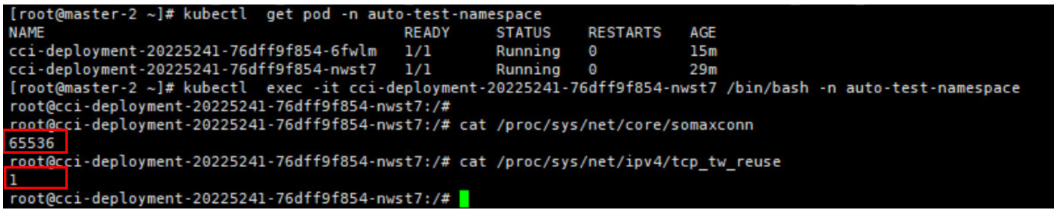
In the following example, the pod's **securityContext** is used to set the sysctl parameters **net.core.somaxconn** and **net.ipv4.tcp_tw_reuse**.

```
apiVersion:v1
kind:Pod
metadata:
    name: xxxxx
    namespace: auto-test-namespace
spec:
   securityContext:
      sysctls:
      - name: net.core.somaxconn
```



```
        value: "65536"
      - name: net.ipv4.tcp_tw_reuse
        value: "1"
      ...
...
```

Go to the container to check whether the configuration takes effect.



```
[root@master-2 ~]# kubectl  get pod -n auto-test-namespace
NAME                                     READY     STATUS     RESTARTS    AGE
cci-deployment-20225241-76dff9f854-6fwlm   1/1       Running    0           15m
cci-deployment-20225241-76dff9f854-nwst7   1/1       Running    0           29m
[root@master-2 ~]# kubectl  exec -it cci-deployment-20225241-76dff9f854-nwst7 /bin/bash -n auto-test-namespace
root@cci-deployment-20225241-76dff9f854-nwst7:/#
root@cci-deployment-20225241-76dff9f854-nwst7:/# cat /proc/sys/net/core/somaxconn
65536
root@cci-deployment-20225241-76dff9f854-nwst7:/# cat /proc/sys/net/ipv4/tcp_tw_reuse
1
root@cci-deployment-20225241-76dff9f854-nwst7:/#
```

3.4 Resizing /dev/shm

Scenario

/dev/shm is a temporary file system (tmpfs), which is a memory-based file system implemented in Linux or Unix and has high read/write efficiency.

If you use **/dev/shm** for data interaction between processes or for temporary data storage, the default size of **/dev/shm** (64 MB) in CCI cannot meet your requirements. CCI allows you to modify the size.

This practice shows how to resize **/dev/shm** by setting memory-backed emptyDir or running **securityContext** and **mount** commands.

Constraints

- **/dev/shm** uses a memory-based tmpfs to temporarily store data. Data is not retained after the container is restarted.
- You can use either of the following methods to modify the size of **/dev/shm**. However, do not use both methods in one pod.
- The emptyDir uses the memory requested by the pod and does not occupy extra resources.
- Writing data to **/dev/shm** is to request memory. In this scenario, you need to evaluate the memory usage of processes. When the sum of the memory requested by processes in the container plus the data volume in the emptyDir exceeds the memory limit of the container, memory overflow occurs.
- When resizing **/dev/shm**, set the size to 50% of the pod's memory request.

Resizing /dev/shm Using Memory-backed emptyDir

emptyDir is applicable to temporary data storage, disaster recovery, and runtime data sharing. It will be deleted upon deletion or transfer of workload pods.

CCI supports the mounting of memory-backed emptyDir. You can specify the memory size allocated to the emptyDir and mount it to the **/dev/shm** directory in the container to resize **/dev/shm**.

```
apiVersion: v1
kind: Pod
metadata:
```

```
 name: pod-emptydir-name
spec:
 containers:
  - image: 'library/ubuntu:latest'
    volumeMounts:
     - name: volume-emptydir1
       mountPath: /dev/shm
    name: container-0
    resources:
     limits:
      cpu: '4'
      memory: 8Gi
     requests:
      cpu: '4'
      memory: 8Gi
 volumes:
  - emptyDir:
     medium: Memory
     sizeLimit: 4Gi
    name: volume-emptydir1
```

After the pod is started, run the **df -h** command to go to the **/dev/shm** directory. If the following information is displayed, the size is successfully modified.

Figure 3-8 /dev/shm directory details

```
root@pod-emptydir-name:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/vdc         20G  182M   19G   1% /
tmpfs            64M     0   64M   0% /dev
tmpfs           4.0G     0  4.0G   0% /sys/fs/cgroup
tmpfs           4.0G   52K  4.0G   1% /etc/hosts
kataShared       20G   45M   19G   1% /dev/termination-log
shm             4.0G     0  4.0G   0% /dev/shm
tmpfs           4.0G     0  4.0G   0% /proc/acpi
tmpfs           4.0G     0  4.0G   0% /proc/scsi
tmpfs           4.0G     0  4.0G   0% /sys/firmware
```

Resizing /dev/shm by Running securityContext and mount Commands

- **Grant the SYS_ADMIN permission to the container.**

Linux provides the SYS_ADMIN permission. To apply this permission to the container, Kubernetes needs to add this information to pods by adding the description of the **securityContext** field to the pod's description file. For example:

```
"securityContext": {
                "capabilities": {
                    "add": [
                        "SYS_ADMIN"
                    ]
                        }
                }
```

Another description field **CapAdd** also needs to be added to the container description.

```
 "CapAdd": [
              "SYS_ADMIN"
         ],
```

In this case, a parameter is added when the container is automatically started by kubelet.

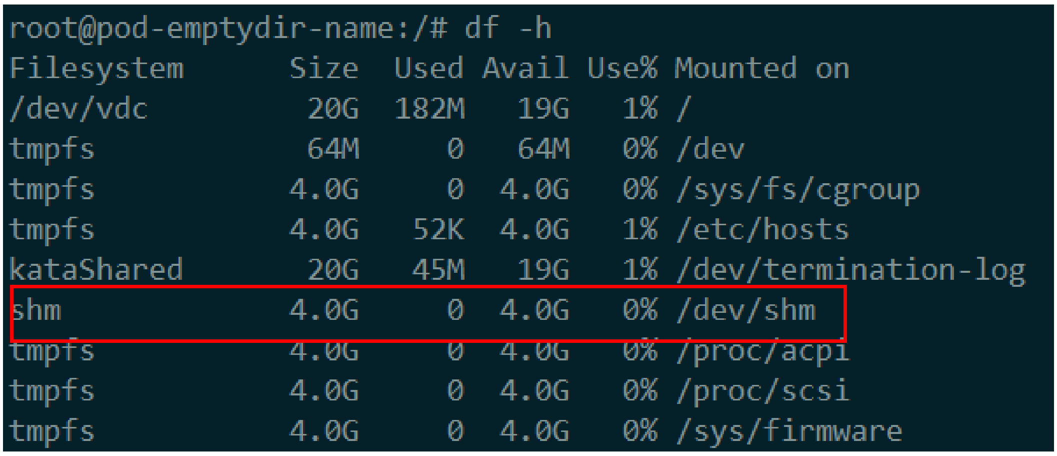
```
docker run --cap-add=SYS_ADMIN
```

- **Insert the mount command in the startup command to resize /dev/shm.**

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-emptydir-name
spec:
  containers:
    - command:
        - /bin/sh
        - '-c'
        - mount -o size=4096M -o remount /dev/shm;bash
      securityContext:
        capabilities:
          add: ["SYS_ADMIN"]
      image: 'library/ubuntu:latest'
      name: container-0
      resources:
        limits:
          cpu: '4'
          memory: 8Gi
        requests:
          cpu: '4'
          memory: 8Gi
```

After the pod is started, run the **df -h** command to go to the **/dev/shm** directory. If the following information is displayed, the size is successfully modified.

Figure 3-9 /dev/shm directory details



```
root@pod-emptydir-name:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/vdc         20G  182M   19G   1% /
tmpfs            64M     0   64M   0% /dev
tmpfs           4.0G     0  4.0G   0% /sys/fs/cgroup
tmpfs           4.0G   52K  4.0G   1% /etc/hosts
kataShared       20G   45M   19G   1% /dev/termination-log
shm             4.0G     0  4.0G   0% /dev/shm
tmpfs           4.0G     0  4.0G   0% /proc/acpi
tmpfs           4.0G     0  4.0G   0% /proc/scsi
tmpfs           4.0G     0  4.0G   0% /sys/firmware
```

