Cloud Container Instance

Best Practice

Issue	01
Date	2020-09-07

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

Notice

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

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

Contents

1 Configuring Dockerfile Parameters for CCI

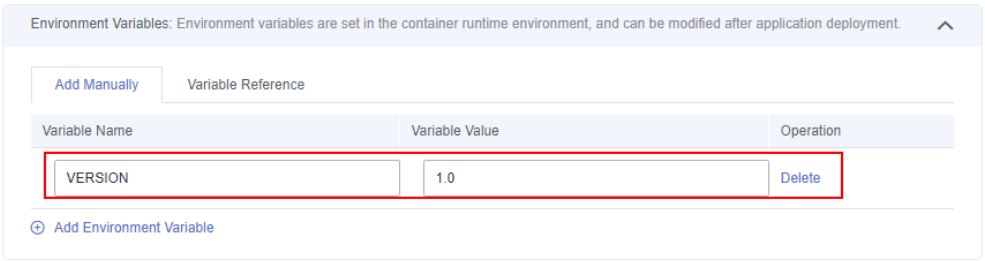
This section describes how to apply Dockerfile configurations to Cloud Container Instance (CCI).

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"]
```

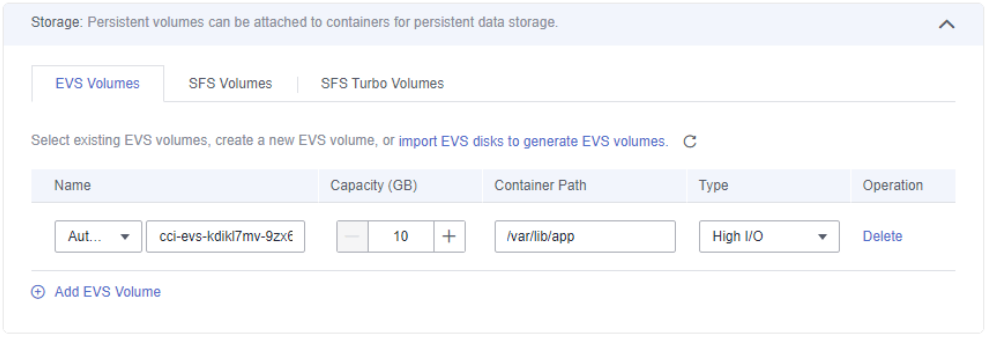
In the preceding example, the Dockerfile contains common parameters, including **ENV**, **VOLUME**, **EXPOSE**, **ENTRYPOINT**, and **CMD**. These parameters can be configured for CCI as follows:

- **ENV** indicates an environment variable. When creating a workload on CCI, set **ENV** in the advanced settings, as shown in the following figure.

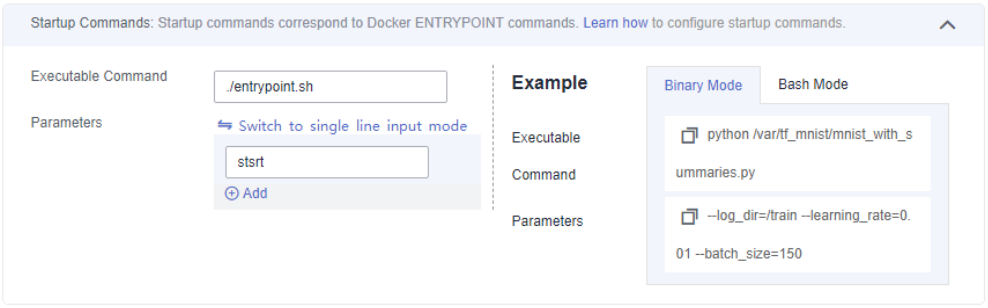
  

- **VOLUME** indicates a container volume. Generally, this parameter is used together with **docker run -v host path:container volume path**.

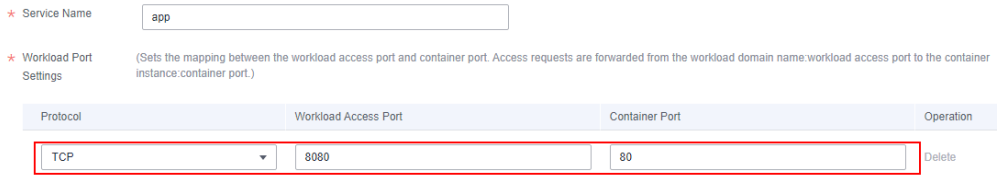
  For CCI, Elastic Volume Service (EVS) disks can be mounted to containers. You only need to add EVS volumes, and configure their sizes and mount paths (that is, container volume paths) when creating workloads.



- **ENTRYPOINT** and **CMD** correspond to the startup command of the advanced setting of CCI. For details, see **Configuring the Container Startup Command**.



- **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**.



2 Creating Workloads with Different Methods

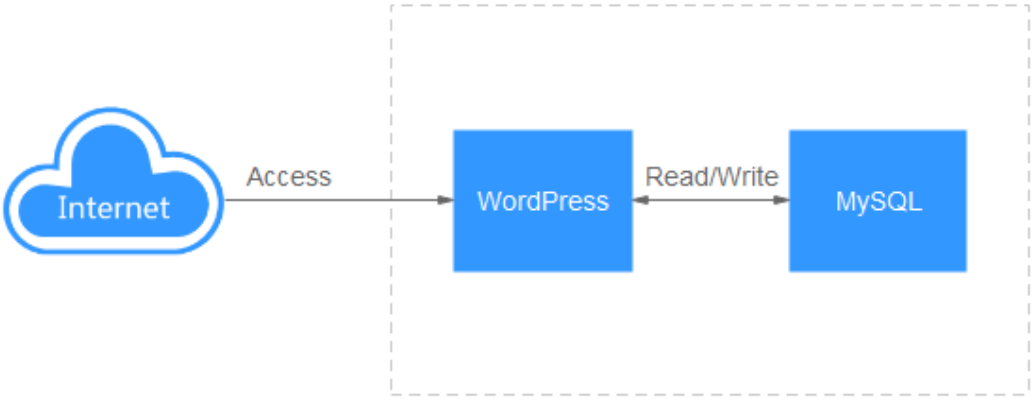
2.1 Overview

On Cloud Container Instance (CCI), you can create workloads by using the console and APIs. What are the differences between the two methods and the method of running the **Docker run** command?

This chapter uses the WordPress and MySQL as examples to illustrate method differences.

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 the MySQL. The WordPress runs the content management program while the 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 Running the Docker run Command to Run Containers

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
```

After the images are downloaded, run the **docker images** command to view the images. Two images exist on the local host, as shown in the following figure.

```
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 directly 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=******** -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 set to **********(replace **********with an actual password).
- **-d**: indicates that the container runs in the backend.

Run the following command to run the WordPress container:

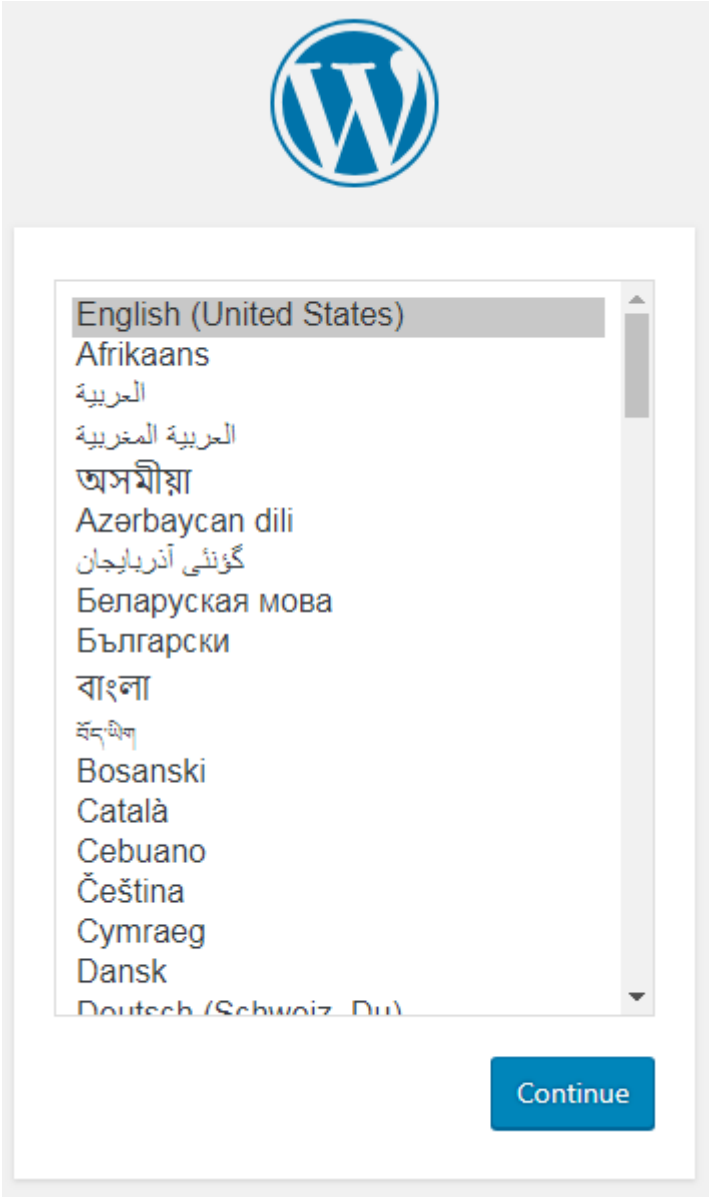
```
docker run --name some-wordpress --link some-mysql:mysql -p 8080:80 -e
WORDPRESS_DB_PASSWORD=******** -d wordpress
```

The following provides the parameter description.

- **--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** is an easy method for connecting 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 set to ******** (replace ******** with an actual password). The value of **WORDPRESS_DB_PASSWORD** must be the same as that of **MYSQL_ROOT_PASSWORD**. This is because the WordPress requires a password to access the MySQL database.
- **-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, as shown in the following figure.



2.3 Using the CCI Console to Create Workloads

Running the Docker run Command to Run Containers describes how to run the WordPress workload by running the **docker run** command. However, it is not convenient to use a container engine in many scenarios, such as auto scaling and rolling upgrade scenarios.

Cloud Container Instance (CCI) provides a serverless container engine, eliminating the need to manage clusters or servers. CCI delivers container agility and high performance with only three steps. CCI enables you to create stateless workloads (Deployments) and stateful workloads (StatefulSets). It 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 Set a VPC.

Select an existing VPC or create one. You must specify a CIDR block for the new VPC. The recommended CIDR blocks 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 CIDR block.

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

Step 6 Click **Create**.

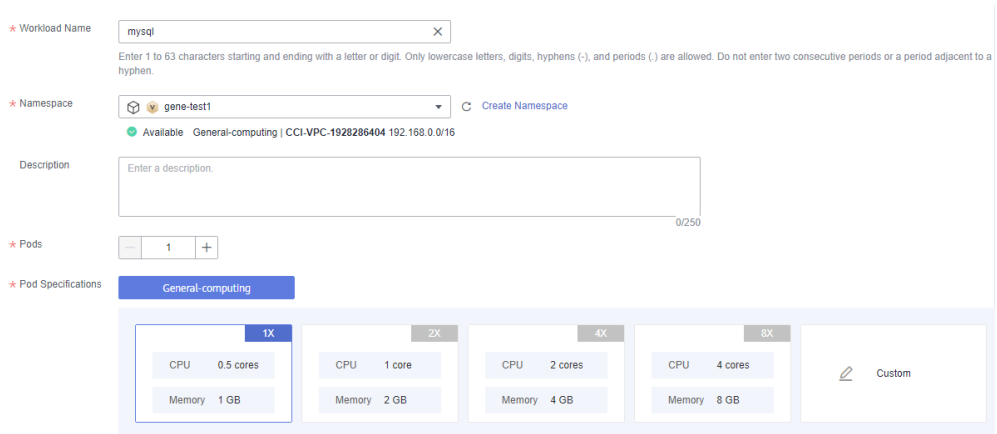
----End

Creating a MySQL Workload

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

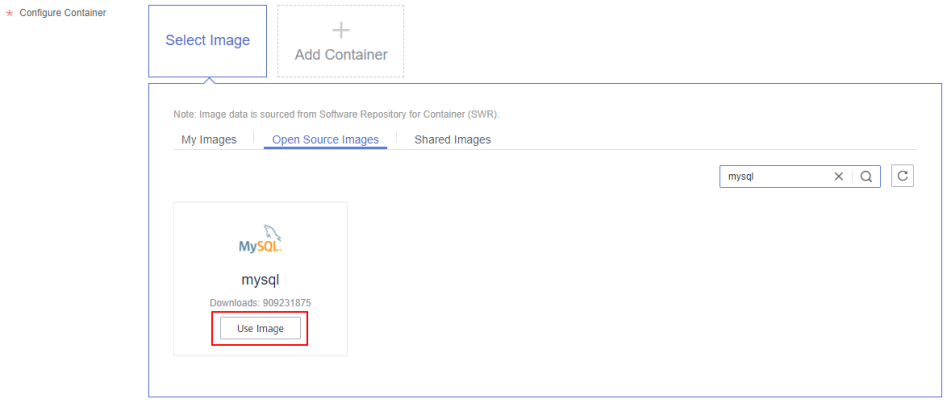
Step 2 Specify basic information.

- **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 GB memory.

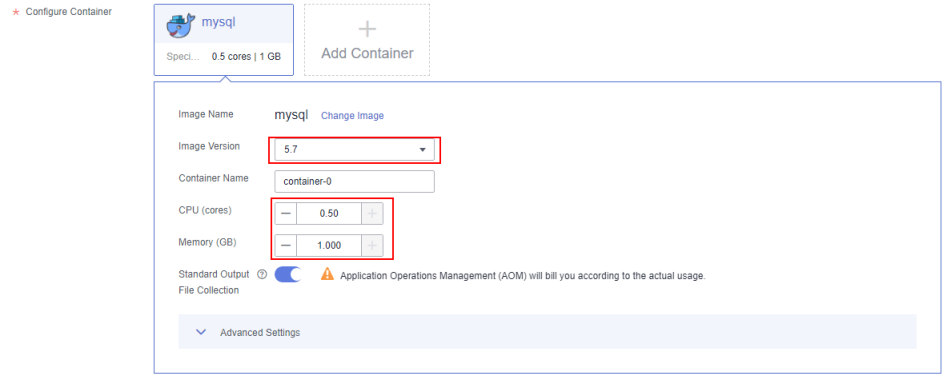


- **Configure Container**

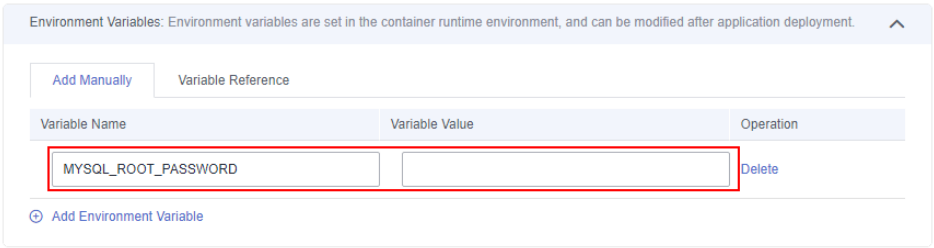
  a. On the **Open Source Images** tab page, search for the mysql image and click **Use Image**.



  b. Set image parameters. Specifically, set the image version to **5.7**, CPU to **0.500** cores, and memory to **1.000** GB.

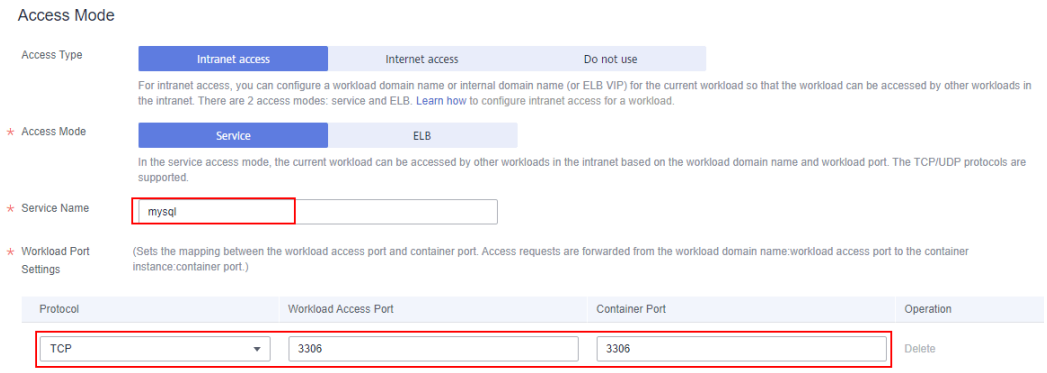


  c. In the advanced settings, enter the environment variable **MYSQL_ROOT_PASSWORD** and its value. The value is the password of the MySQL database. You need to set the password by yourself.



Step 3 Click **Next** and configure the workload information. 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 After the configuration is complete, 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.

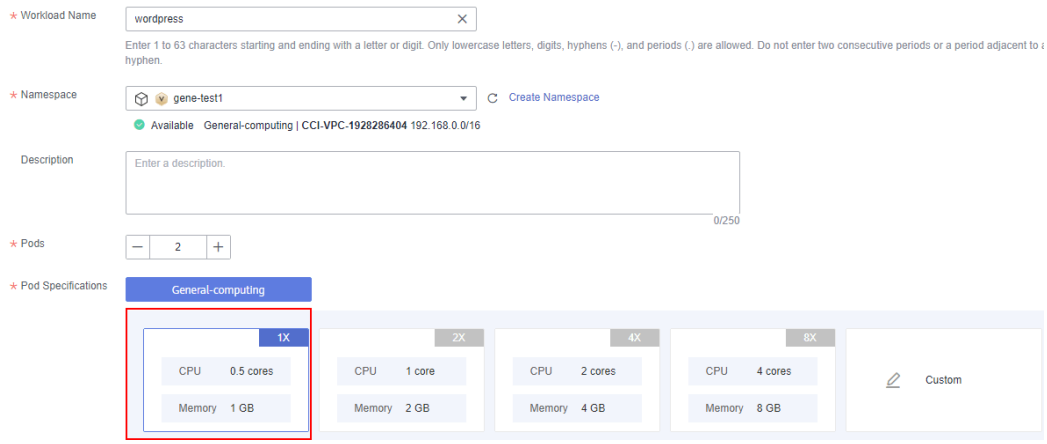
----End

Creating a WordPress Workload

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

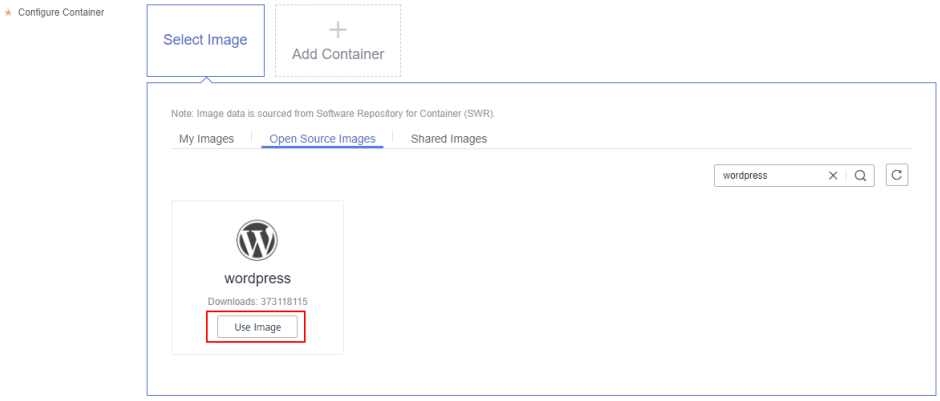
Step 2 Specify basic information.

- **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 GB memory.

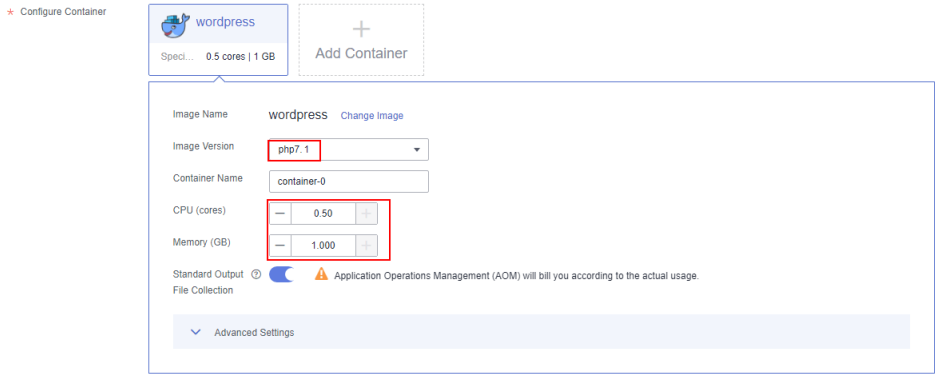


- **Configure Container**

  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 **0.500** cores, and memory to **1.000** GB.



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

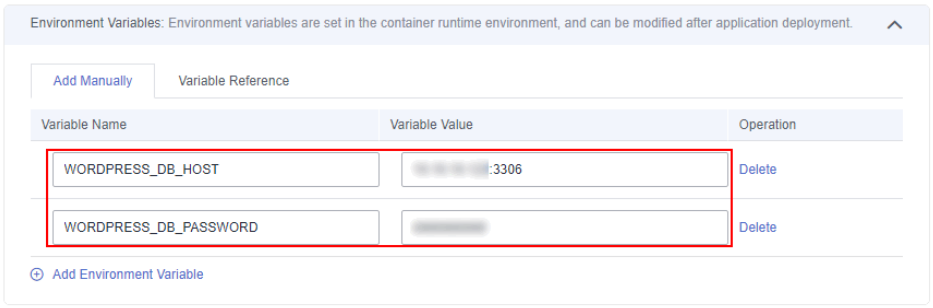
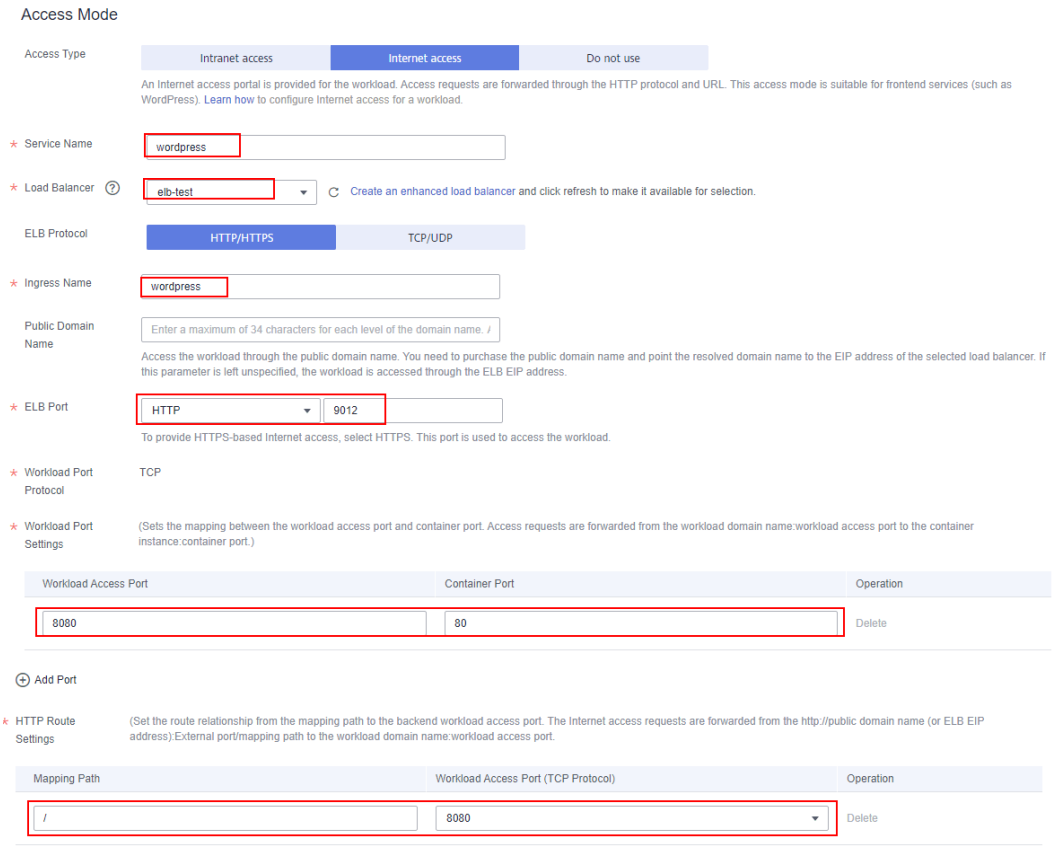


Table 2-1 Description of environment variables

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** and configure the workload information.

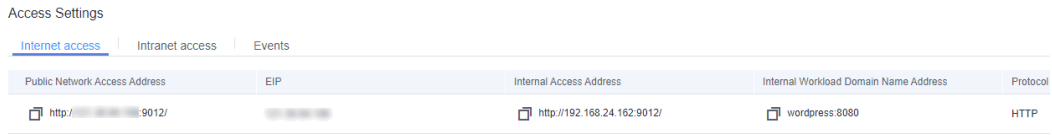
Set **Access Type** to **Internet access** and **Service Name** to **wordpress**, and select a load balancer. If no load balancers are available, click **Create an enhanced load balancer** to create one. Then, set **ELB Protocol** to **HTTP** and **ELB Port** to **9012**. In the **Workload Port Settings** area, set a mapping between workload access port 8080 and container port 80 (default access port of the WordPress image). In the **HTTP Route Settings** area, set **Mapping Path** to **/** (that is, use **http://Load balancer IP address:External port** to access the WordPress) and **Workload Access Port** to **8080**.



Step 4 After the configuration is complete, 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, that is, **Load balancer IP address:port**.



----End

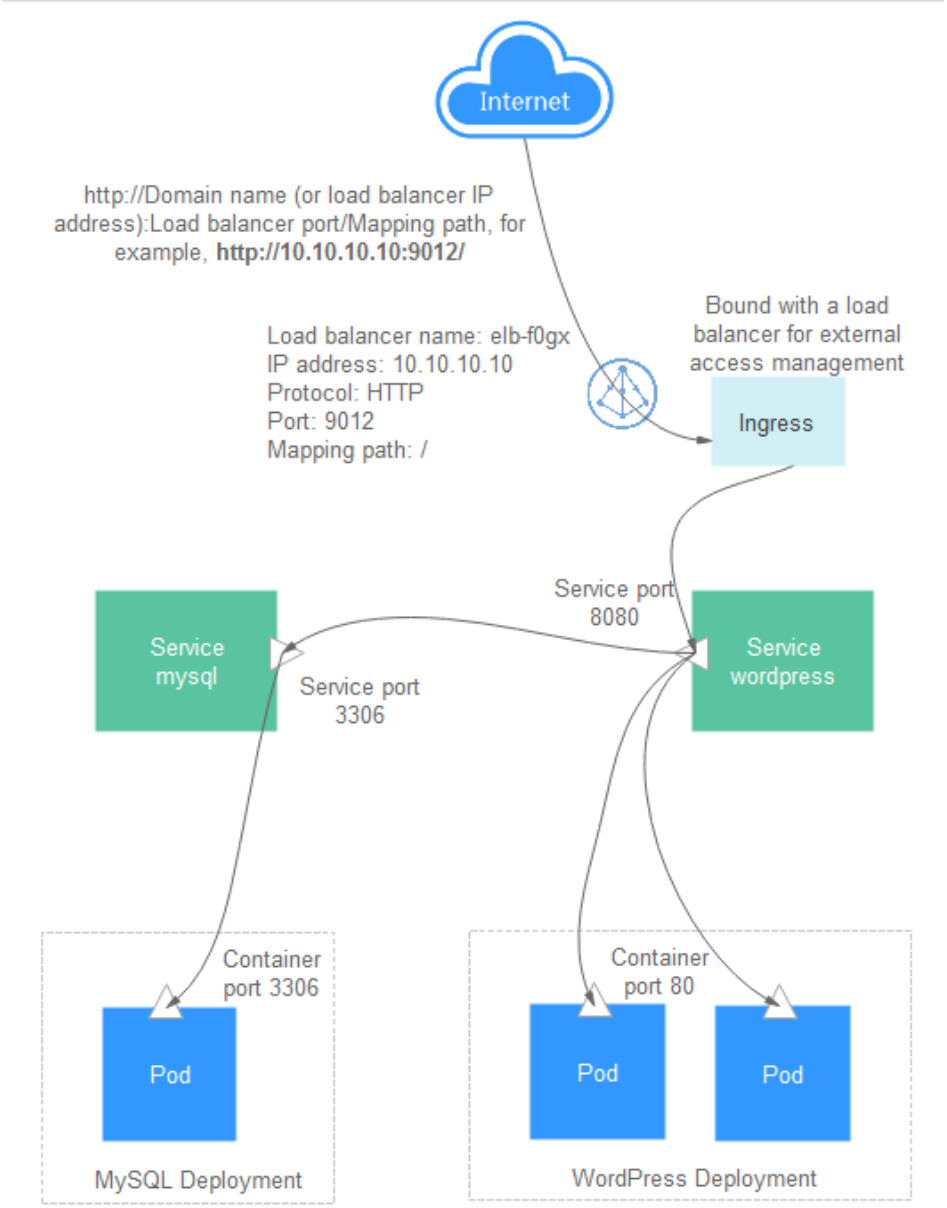
2.4 Calling APIs to Create Workloads

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 ***CCI Development 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 in **Creating a Namespace** to create a namespace and specify a namespace type.

```
curl POST https://cci.cn-north-1.myhuaweicloud.com/api/v1/namespaces -H 'content-type: application/json' -H "x-auth-token: $Token" -v -k -d '{
   "apiVersion": "v1",
   "kind": "Namespace",
   "metadata": {
      "name": "namespace-test",
      "annotations": {
         "namespace.kubernetes.io/flavor": "gpu-accelerated"
      }
   },
   "spec": {
      "finalizers": [
         "kubernetes"
      ]
```



```
    }
}'
```

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

```
curl POST https://cci.cn-north-1.myhuaweicloud.com/apis/networking.cci.io/v1beta1/namespaces/namespace-test/networks -H 'content-type: application/json' -H "x-auth-token: $Token" -v -k -d '{
    "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": "cnnorth1a",
        "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 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 an actual 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 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**, that is, associate the pod created in **Step 1**.
- Map workload access port 3306 to container port 3306.
- Set the access type of the service to **NodePort**. That is, resources are accessed through the port of the node where the pod is located.

```
{
  "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 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 an actual 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 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**, that is, 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.
- Set the access type of the service to **NodePort**. That is, resources are accessed through the port of the node where the pod is located.

```
{
   "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 in **Creating an Ingress** to create an ingress to define the external access policy of the WordPress. That is, associate a load balancer (it must be in the same VPC as the WordPress workload).

- metadata.annotations.kubernetes.io/elb.id: must be data.
- metadata.annotations.kubernetes.io/elb.ip: IP address of the load balancer.
- metadata.annotations.kubernetes.io/elb.port: port of the load balancer.
- spec.rules: set of rules for accessing the service. A path list is in the format like "/". 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 ELB 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

3 Using TensorFlow to Train Neural Networks

This topic describes how to create a GPU-accelerated workload in CCI and how to use such a workload to train a simple neural network in a container. The TensorFlow image classification is used as an example.

Such container-based AI training and inference have the following advantages:

- Environment differences can be eliminated. You do not need to install various software, such as Python, TensorFlow, and CUDA toolkit.
- The GPU driver is installation-free.
- The resource cost is low, and resources are charged by second.
- The serverless architecture does not require VM O&M.

Creating an Image

The TensorFlow community provides the base TensorFlow images that are installed with the base TensorFlow library. TensorFlow images are classified into GPU-enabled images and CPU-enabled images, which can be downloaded from the following addresses:

- GPU-enabled images: tensorflow/tensorflow:1.15.0-gpu
- CPU-enabled images: tensorflow/tensorflow:1.13.0

In this example, a trained model named Inception-v3 from the TensorFlow official website is used to classify images. Inception-v3 is a model trained in the 2012 ImageNet Challenge. In this challenge, it classified a huge image set into 1000 types. GitHub provides the code for classifying images by using Inception-v3.

The code for training models is contained in the project **https://gpu-demo.obs.cn-north-1.myhuaweicloud.com/gpu-demo.zip**. You need to download and decompress the package, and add the code project to an image. The content of the Dockerfile for creating an image is as follows:

```
FROM tensorflow/tensorflow:1.15.0-gpu
ADD gpu-demo /home/project/gpu-demo
```

The preceding **ADD** command is used to copy the gpu-demo project to the **/home/project** directory of the image. You can modify the directory as required.

Run the **docker build -t tensorflow/tensorflow:v1 .** command to create an image. The dot (.) indicates the current directory, that is, the directory where the Dockerfile is located.

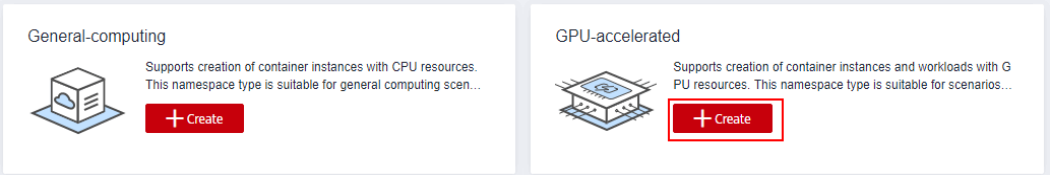
After the image is created, push it to the SoftWare Repository for Container (SWR). For details about how to push an image, see **Introduction**.

Creating a TensorFlow Workload

Step 1 Log in to the CCI console.

Step 2 In the navigation pane, choose **Namespaces**. On the page displayed on the right, click **Create** in the GPU-accelerated area. In the displayed dialog box, enter the namespace name, set the VPC and subnet CIDR blocks, and click **Create**.

Figure 3-1 GPU-accelerated namespace



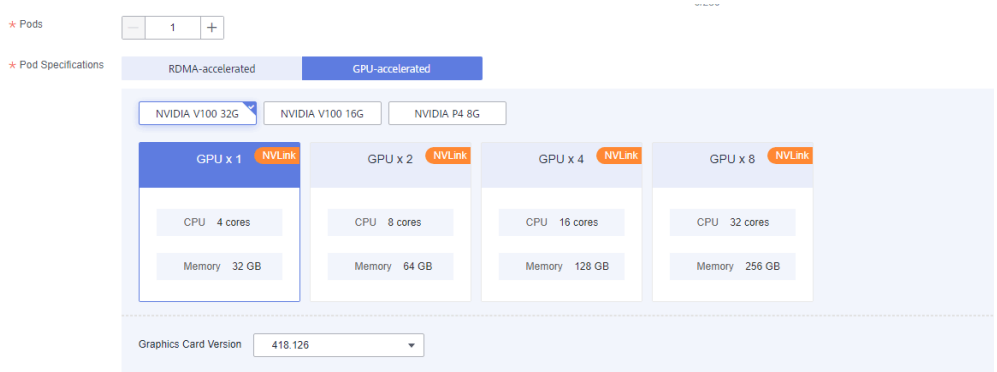
Step 3 In the navigation pane, choose **Workloads** > **Deployments**. On the page displayed on the right, click **Create Deployment**.

Step 4 Configure workload information.

1. Specify the workload name, select the namespace created in **Step 2**, set **Pods** to **1**, and select **GPU-accelerated** for **Pod Specifications** and **418.126** for the GPU driver version.

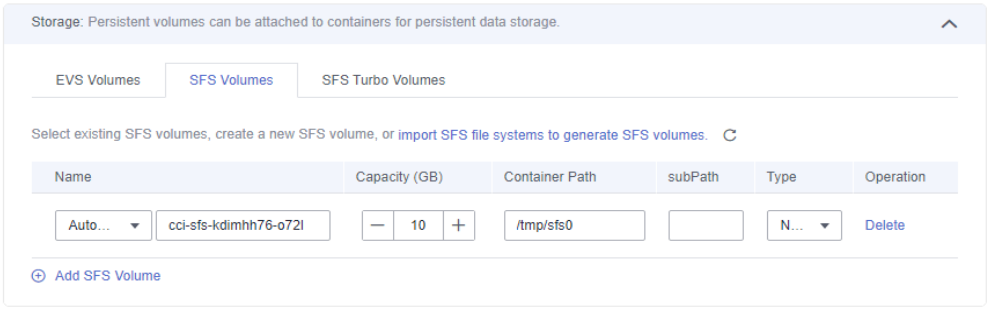
   For details about GPU-accelerated pod specifications and GPU drivers, see **Pod Specifications**.

   Figure 3-2 Selecting GPU-accelerated pod specifications



2. Select an image. In this example, select the TensorFlow image pushed to SWR.
3. In the **Advanced Settings** area, mount an SFS volume of the NFS type to store the trained data.

Figure 3-3 Mounting NFS-type volume



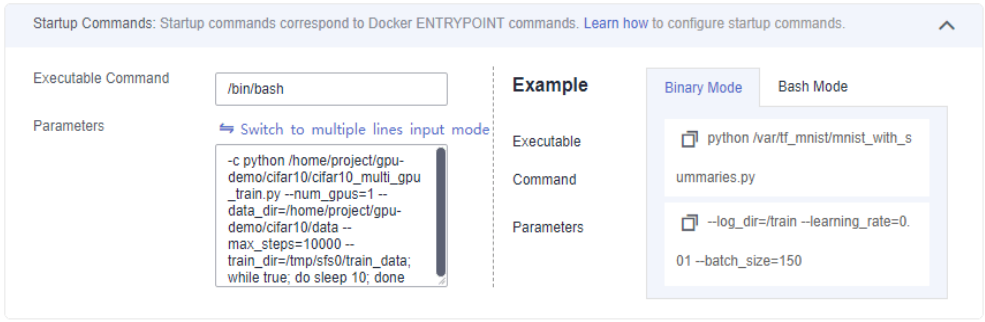
4. In the **Startup Commands** area, enter an executable command and parameters.
   - Executable command: /bin/bash
   - Parameter 1: -c
   - Parameter 2: python /home/project/gpu-demo/cifar10/cifar10_multi_gpu_train.py --num_gpus=1 --data_dir=/home/project/gpu-demo/cifar10/data --max_steps=10000 --train_dir=/tmp/sfs0/train_data; while true; do sleep 10; done

     --train_dir indicates the path for storing the training result. The path prefix **/tmp/sfs0** must be the same as **Container Path** specified in **Step 4.3**. Otherwise, the training result cannot be written into the NFS-type volume.

     --max_steps indicates the number of training iterations. In this example, this field is set to **10000**. The model training takes about 3 minutes. If this parameter is not specified, the default value **1000000** is used, which indicates that the model training takes a longer time. Increasing the value of **max_steps** will result in a longer training time and a more accurate result.

   The preceding command is used to train the image classification model. After setting the command and parameters, click **Next**.

Figure 3-4 Setting the container startup command



5. Configure workload access settings.

   In this example, select **Do not use**. Then, click **Next**.

6. Click **Submit**, and then click **Back to Deployment List**.

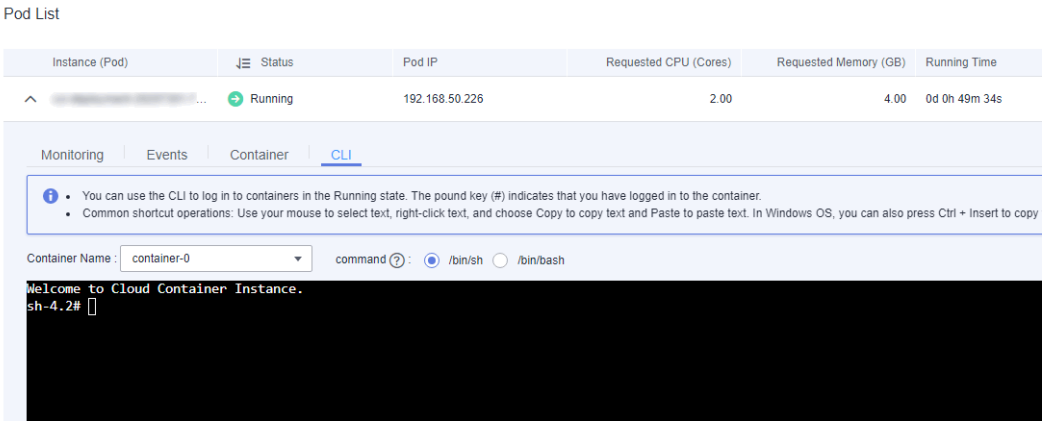
   In the workload list, if the workload is in the **Running** state, the workload is successfully created.

----End

Using an Existing Model to Classify Images

Step 1 Click the TensorFlow workload name. In the **Pod List** area of the workload details page, click the arrow icon at the left of the pod and then click the **CLI** tab. If the prompt (#) is displayed on the CLI, you have logged in to the pod.

Figure 3-5 Accessing the pod using the web-terminal



Step 2 Switch to the directory where the project is located, and run the **python classify_image.py --model_dir=model** command to query the classification result.

```
# cd /home/project/gpu-demo
# ls -l
total 96
-rw-r--r-- 1 root root  6874 Aug 30 08:09 airplane.jpg
drwxr-xr-x 3 root root  4096 Sep  4 07:54 cifar10
drwxr-xr-x 3 root root  4096 Aug 30 08:09 cifar10_estimator
-rw-r--r-- 1 root root 30836 Aug 30 08:09 dog.jpg
-rw-r--r-- 1 root root 43675 Aug 30 08:09 flower.jpg
drwxr-xr-x 4 root root  4096 Sep  4 02:14 inception
# cd inception
# python classify_image.py --model_dir=model --image_file=/home/project/gpu-demo/
airplane.jpg
...
2019-01-02 08:05:24.891201: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1084] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15131 MB memory) -> physical GPU
(device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:0a.0, compute capability: 6.0)
airliner (score = 0.84250)
wing (score = 0.03228)
space shuttle (score = 0.02524)
warplane, military plane (score = 0.00691)
airship, dirigible (score = 0.00664)
```

In the preceding command, **--image_file** specifies the image (as shown in the following figure) to be classified. The last lines of the result are the classification label and the corresponding score. A higher score indicates a more accurate classification. The line **airliner (score = 0.84250)** indicates that the model recognizes the image as an airliner.

Figure 3-6 airliner

You can also do not specify the image to be classified. If an image is not specified, the following image is used.

Figure 3-7 Default image

In this case, run the **python classify_image.py –model_dir=mode** command to query the classification result.

```
# python classify_image.py --model_dir=model
...
2019-01-02 08:02:33.271527: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1084] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15131 MB memory) -> physical GPU
(device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:0a.0, compute capability:
6.0)
giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (score =
0.89107)
indri, indris, Indri indri, Indri brevicaudatus (score = 0.00779)
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score =
0.00296)
custard apple (score = 0.00147)
earthstar (score = 0.00117)
```

The result shows that the model recognizes the image as a panda.

----End

Using the Trained Image Classification Model

The TensorFlow official website provides the model code and training data of a Deep Convolutional Neural Network (DCNN), that is, CIFAR-10. CIFAR-10 is a simplified image classification model. It classifies images across 10 categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The images trained in the model, also called training data, are of these 10 categories.

Step 1 On the CCI console, click the workload name. In the **Pod List** area of the workload details page, click the arrow icon at the left of the pod and then click the **CLI** tab. Then, use **cifar10_eval.py** provided in the code to check the accuracy of the model. In the following command, set the **checkpoint_dir** field to the directory where the model that has just been trained is located.

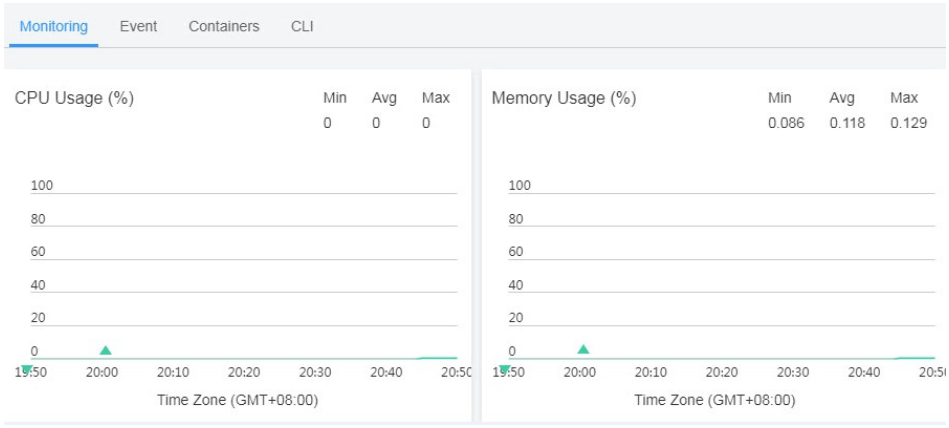

```
# cd /home/project/gpu-demo/cifar10
# python cifar10_eval.py --data_dir=data --checkpoint_dir=/tmp/sfs0/train_data --run_once
...
2019-01-02 08:25:43.914186: precision @1 = 0.817
```

Step 2 Continue to use the preceding airplane image for testing. In the following command, set the **checkpoint_dir** field to the directory where the model that has just been trained is located and the **test_file** field to the image to be tested.

```
# python label_image.py --checkpoint_dir=/tmp/sfs0/train_data --test_file=/home/project/gpu-demo/
airplane.jpg
 ...
2019-01-02 08:36:42.149700: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1084] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15131 MB memory) -> physical GPU
(device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:0a.0, compute capability:
6.0)
airplane (score = 4.28143)
ship (score = 1.92319)
cat (score = 0.03095)
```

The result shows that the model accurately recognizes the image as an airplane. **label_image.py** is the code that uses the model that has just been trained to classify the image.

In addition, you can view the usage of various resources on the **Monitoring** tab page in the **Pod List** area.



----End