

DataArtsFabric

Best Practices

Issue 01
Date 2025-07-18



Copyright © Huawei Cloud Computing Technologies Co., Ltd. 2025. 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.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: <https://www.huaweicloud.com/intl/en-us/>

Contents

1 Using Ray for Small Model Inference..... 1

2 Example of Configuring Permissions for Using the Inference Service..... 6

3 Using DataArts Fabric SQL to Query Data..... 9

1 Using Ray for Small Model Inference

Description

Small model inference indicates the process of performing inference on relatively small machine learning models. These models may be able to run efficiently on a single compute node due to lower complexity and fewer parameters. Even small models can struggle with large data volumes. To mitigate this, Ray can be employed for parallel and distributed inference to improve inference performance.

Ray is a fully managed service in DataArts Fabric that makes it easy to manage small model inference. You only need to define your small model inference process as an executable task of Ray and create and run an inference job in DataArts Fabric.

Prerequisites

- You have [created an OBS bucket](#).
- You have [created a DataArts Fabric workspace](#).
- You have [created a DataArts Fabric Ray resource](#).
- You have [created a DataArts Fabric Ray cluster](#).

Step 1: Prepare a Code Script

The following Python code script shows how to create an inference job, perform simple linear regression model inference, and use the distributed scheduling capability of Ray to calculate the inference result. The script is only for reference and you can create your own script for subsequent inference tasks as needed.

- **simple_model.py** is used to define and start a model. The script defines a linear regression model **SimpleModel** and a model deployment **serve**.

```
# simple_model.py
from sklearn.linear_model import LinearRegression
import numpy as np
import pickle
import ray
from ray import serve
from fastapi import FastAPI, Request
from ray.serve.handle import DeploymentHandle

app = FastAPI()
```

```
class SimpleModel:
    def __init__(self):
        self.model = LinearRegression()
        X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
        y = np.dot(X, np.array([1, 2])) + 3
        self.model.fit(X, y)
    def predict(self, X):
        return self.model.predict(X).tolist()

model_instance = SimpleModel()

@serve.deployment(name="simple_model_deployment", ray_actor_options={"num_cpus": 1})
@serve.ingress(app)
class SimpleModelDeployment:
    def __init__(self, model: DeploymentHandle):
        self.model = model
    @app.post("/predict")
    async def predict(self, request: Request):
        request_data = await request.json()
        input_data = np.array(request_data).reshape(-1, 2)
        prediction = self.model.predict(input_data)
        return {"prediction": prediction}

deployment_instance = SimpleModelDeployment.bind(model=model_instance)
serve.run(deployment_instance)
```

- **infer_client.py** indicates the main entry script for invoking the client. The main process includes calling the model script to deploy the model, inputting the input data into the model for inference, and uploading the inference results to OBS. The **-ak**, **-sk**, **-ep**, and **-dp** parameters must be specified in the script.
 - **-ak**: AK of OBS. To obtain the value, see [Where Can I Obtain Access Keys \(AK and SK\)?](#)
 - **-sk**: SK of OBS. To obtain the value, see [Where Can I Obtain Access Keys \(AK and SK\)?](#)
 - **-ep**: OBS endpoint. To obtain the value, see [Endpoints and Domain Names](#).
 - **-dp**: OBS path and file where inference output files are stored.

```
# client.py
import requests
import numpy as np
from obs import ObsClient
from urllib.parse import urlparse
import argparse
from dataclasses import dataclass
import os
from ray import serve
import subprocess
import multiprocessing
import time
import ray
input_file_path = './input.txt'
output_file_path = './output.txt'

def run_model():
    # serve run simple_model:deployment_instance
    subprocess.run(['python3', './simple_model.py'])
@dataclass(frozen=True)
class ParsedObsPath:
    bucket_id: str
    key_id: str
def do_inference():
    input_data = []
    with open(input_file_path, 'r') as f:
        for line in f:
            parts = line.strip().split()
```


```
        input_data.append([float(parts[0]), float(parts[1])])
input_data_array = np.array(input_data).tolist()
print(f"input_data_array={input_data_array}")
response = requests.post("http://localhost:8000/predict", json=input_data_array)
print(f"response: {response}")
predictions = response.json()["prediction"]
with open(output_file_path, 'w') as f:
    for prediction in predictions:
        f.write(f"{prediction}\n")
print(f"result save in: {output_file_path}")
def parse_obs_uri(path: str) -> ParsedObsPath:
    if not path.startswith('obs://'):
        raise Exception(f'OBS path format incorrect: "{path}"')
    parsed = urlparse(path)
    return ParsedObsPath(bucket_id=parsed.netloc, key_id=parsed.path[1:])
def upload_file_to_obs(obs_client: ObsClient, obs_path: str, source_path: str):
    if not os.path.exists(source_path):
        raise Exception(
            f'Source file is not exist: source_path={source_path}')
    uri = parse_obs_uri(obs_path)
    # ObsClient.putFile(bucketName, objectKey, file_path, metadata, headers, progressCallback)
    print(f"bucket_id={uri.bucket_id}, key_id={uri.key_id}, source_path={source_path}")
    result = obs_client.putFile(
        bucketName=uri.bucket_id,
        objectKey=uri.key_id,
        file_path=source_path
    )
    return result
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("-ak", "--access_key_id", help="OBS access key",
                        type=str, required=True)
    parser.add_argument("-sk", "--secret_access_key", help="OBS secret key",
                        type=str, required=True)
    parser.add_argument("-st", "--security_token", help="OBS security token",
                        type=str, required=False)
    parser.add_argument("-ep", "--endpoint", help="OBS endpoint", type=str,
                        required=True)
    parser.add_argument("-dp", "--dst_path",
                        help="Local filesystem destination path", type=str,
                        required=True)
    args = parser.parse_args()
    obs_client = ObsClient(
        access_key_id=args.access_key_id,
        secret_access_key=args.secret_access_key,
        security_token=args.security_token,
        server=args.endpoint,
        signature='obs'
    )
    # run model
    print("start run model")
    background_process = multiprocessing.Process(target=run_model)
    background_process.start()
    # wait ray serve ready
    time.sleep(20)
    # do inference
    print("start do inference")
    do_inference()
    # upload infer result to obs
    print("start upload result to obs")
    upload_result = upload_file_to_obs(
        obs_client=obs_client,
        obs_path=args.dst_path,
        source_path=output_file_path
    )
    print(f"upload result={upload_result}")
    if not upload_result.status < 300:
        raise Exception('Error while uploading to OBS.'
                        f' upload status: {upload_result.status}')
```

```
ray.shutdown()
if __name__ == "__main__":
    main()
```

- **input.txt** indicates what need to be inferred.

```
3.0 5.0
1.0 2.0
4.0 6.0
2.0 3.0
```

Step 2: Upload the Code Script to an OBS Bucket

1. Log in to the Huawei Cloud console, click  in the upper left corner, and choose **Storage > Object Storage Service**.
2. Upload the code script created in [Step 1](#) to the OBS bucket. For details, see [Simple Upload \(PUT\)](#).
3. After the script is uploaded, check the uploaded script in the OBS bucket. You can select the script when creating a Ray job.

For example, upload the job script to the **obs://fabric-job-test/rayJob/ray-job/RayInferDemo2** directory.

Step 3: Create an Inference Job

1. Log in to DataArts Fabric Workspace Management Console, select the created workspace, and click **Access Workspace**.
2. In the navigation pane, choose **Development and Production > Jobs**. Click **Create Job** in the upper right corner. For details, see [Creating a Ray Job](#).

Table 1-1 Parameter description

Parameter	Description
Code Directory	Select the path uploaded in Step 2 , for example, obs://fabric-job-test/rayJob/ray-job/RayInferDemo2 .
Ray Main File	Select the main entry script of the entire job, for example, infer_client.py .
Ray Job Parameters	Enter the parameters required for executing the main entry script. For details about how to obtain the AK and SK, see Where Can I Obtain Access Keys (AK and SK)? For details about how to obtain the endpoint, see Endpoints and Domain Names . Example: -ak XXXXXXXXXXXXXXXX -sk xxxxxxxxxxxxxxxx -ep obs.cn-north-7.huawei.com -dp obs://fabric-job-test/test_output/output.txt
Dependencies	Software and its version on which the Ray job depends. If there are multiple dependencies, enter them in different lines. Example: scikit-learn==1.5.2 numpy==1.19.5

Step 4: Run a Job

1. After the job is defined, ensure that an available Ray cluster is selected for the job. Locate the target job in the job list and click **Start** in the **Operation** column.
2. Locate the target job in the job list and click **View Details** in the **Operation** column. Check the job status in the **Run** tab.

You can view the output results in the OBS bucket path.

2 Example of Configuring Permissions for Using the Inference Service

Description

Assume that a company needs to use the DataArts Fabric service and has the following requirements:

- As the IAM permission administrator, Tom needs to authorize services on the DataArts Fabric page and configure IAM permissions for different roles.
- As a development engineer, John needs to create a workspace and a Ray cluster.
- As an algorithm engineer, Robert needs to use the inference file stored in OBS to create a personal inference model or uses the public inference model for experiments.
- As a test engineer, William needs to query and use the work achievements of other colleagues.

Prerequisites

- They all have available Huawei Cloud accounts.
- Tom has the IAM administrator and DataArtsFabricFullPolicy permissions.

Identity and Permission Description

Table 2-1 Identity and permission description

Employee	Identity	Permissions
Tom	IAM permission administrator	DataArtsFabricFullPolicy and IAM Agency Management FullAccess (used to create an agency for DataArts Fabric) are required.

Employee	Identity	Permissions
John	Development engineer	<ul style="list-style-type: none">• DataArtsFabricFullPolicy is required to create workspaces. To specify LakeFormation Metastore during the workspace creation, LakeFormation ReadOnly Access is also required.• To create a Ray cluster, DataArtsFabricFullPolicy includes only the permission to create an order for purchasing Ray resources, but does not include the payment permission. Payment must be performed by a fee administrator specified by the customer.
Robert	Algorithm engineer	DataArtsFabricFullPolicy and required OBS permissions are required. To use model files in OBS in DataArts Fabric, some OBS permissions need to be granted by the user permission administrator Tom.
William	Test engineer	DataArtsFabricReadOnlyPolicy policy is required to perform read-only operations.

Procedure

- Step 1** Tom: Log in to the DataArts Fabric console and grant permissions. For details, see [Creating an IAM User and Assigning Permissions to Use DataArts Fabric](#).
- Grant the DataArtsFabricFullPolicy and LakeFormation ReadOnly Access permissions to John.
 - Grant the DataArtsFabricFullPolicy permissions and OBS OperateAccess permissions for the OBS bucket **my-obs-bucket** to Robert.
 - Grant the DataArtsFabricReadOnlyPolicy permissions to William.
- Step 2** John: Create a workspace on the DataArts Fabric console and specify MetaStore. For details, see [Creating a Workspace](#).
- Step 3** John: In the new workspace, purchase Ray resources and generate an order. For details, see [Purchasing a Ray Resource](#).
- Step 4** Tom: Go to the order payment page and pays for the order. After the payment is complete, the Ray cluster is automatically created. For details, see [Purchasing a Ray Resource](#).
- Step 5** John: Execute a job on the Ray cluster. For details, see [Creating a Ray Job](#).
- Step 6** Robert: Create a model on the **Model** page and specify the OBS address of the model file. For details, see [Creating a Model](#).
- Step 7** Robert: Use the model to build an inference service and completes the test and debugging in the playground. For details, see [Using an Inference Service for Inference](#).

Step 8 William: Check the running result of the Ray job and tests the inference service built by Robert in the playground. For details, see [Managing Ray Jobs](#) and [Performing Inference in the Playground](#).

----End

3 Using DataArts Fabric SQL to Query Data

Description

DataArts Fabric SQL is a cloud-native serverless version of DataArts Fabric. By leveraging the resource pooling and massive storage capabilities provided by the cloud infrastructure, it combines parallel execution, metadata decoupling, and compute-storage persistent decoupling architecture for robust elasticity and lake warehouse integration.


This section describes how to quickly enable DataArts Fabric SQL and perform simple data queries.

Prerequisites

- You have registered an account and completed real-name authentication. The account is not in arrears or frozen.
- You have enabled LakeFormation and OBS permissions and confirmed the agency.
- You have a workspace available.

Step 1: Plan and Create an OBS Bucket and Import Data


DataArts Fabric SQL uses OBS to store data. You need to create a bucket and folder on the OBS console and import sample data.

1. Log in to the management console.
2. Click  in the upper left corner of the page and choose **Storage > Object Storage Service** to access the **Object Storage Service** console.
3. Use a parallel file system as an example.
Click **Parallel File Systems** and click **Create Parallel File System**. On the displayed page, set the parameters, and click **Create Now**.
 - Configure **File System Name** as required, for example, to **fabric-serverless**.
 - Configure other parameters based on site requirements.


4. On the **Parallel File System** page, click the name of the created file system, for example, **fabric-serverless**.
5. Click **Files** in the navigation pane, click **Create Folder**, enter a folder name, and click **OK**. Click the folder name and click **Create Folder** to create a subfolder.
6. Repeat this step to create paths for storing metadata in sequence. The following paths are examples:
 - Catalog storage path: **fabric-serverless/catalog1**
 - Database storage path: **fabric-serverless/catalog1/database1**
 - Data table storage path: **fabric-serverless/catalog1/database1/table1**

Step 2: Plan and Create a LakeFormation Instance, Catalog, and Database

DataArts Fabric SQL manages data sources using LakeFormation. You need to purchase a LakeFormation instance and configure its catalog, database, and table information.

1. Log in to the management console.
2. In the upper left corner of the page, choose **Analytics > DataArts Lake Formation**.
3. On the **Overview** page, purchase an instance.
4. In the upper left corner, select the instance to display its details.
5. Create a catalog.
 - a. In the navigation pane on the left, choose **Metadata > Catalog**.
 - b. Click **Create**. On the displayed page, configure the following parameters, and click **Submit**.
 - **Catalog Name:** **catalog1**
 - **Select Location:** Click , select a storage location, for example, **obs://fabric-serverless/catalog1**, and click **OK**.
 - **Catalog Type:** **DEFAULT**
 - Retain the default settings for other parameters.
 - c. After the catalog is created, check the catalog information on the **Catalog** page.
6. Create a database.
 - a. In the navigation pane on the left, choose **Metadata > Database**.
 - b. Select **catalog1** from the drop-down list box next to **Catalog** in the upper right corner.

If the database **default** already exists, skip this step.
 - c. Click **Create**, configure related parameters, and click **Submit**.
 - **Database Name:** **database1**
 - **Catalog:** **catalog1**

- **Select Location:** Click , select a location, for example, **obs://fabric-serverless/catalog1/database1**, and click **OK**.
 - Retain the default settings for other parameters.
- d. After the database is created, check the database information on the **Database** page.

Step 3: Use DataArts Fabric SQL

1. Log in to the Huawei Cloud DataArts Fabric console and click **Access Workspace**.
2. In the navigation pane on the left, choose **Development and Production > SQL Editor**. Select a LakeFormation instance and a LakeFormation catalog.
3. Select an SQL endpoint to run the SQL statements.

```
CREATE TABLE
database1.iceberg_table (
  col_id INT,
  col_tinyint SMALLINT,
  col_smallint SMALLINT,
  col_int INTEGER,
  col_bigint BIGINT
) store AS iceberg;

SELECT * FROM database1.iceberg_table;
```