

ModelArts

Best Practices

Issue 01
Date 2024-06-12



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

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

Trademarks and Permissions



HUAWEI 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.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: <https://www.huawei.com>

Email: support@huawei.com

Security Declaration

Vulnerability

Huawei's regulations on product vulnerability management are subject to the *Vul. Response Process*. For details about this process, visit the following web page:

<https://www.huawei.com/en/psirt/vul-response-process>

For vulnerability information, enterprise customers can visit the following web page:

<https://securitybulletin.huawei.com/enterprise/en/security-advisory>

Contents

1 Permissions Management.....	1
1.1 Basic Concepts.....	1
1.2 Permission Management Mechanisms.....	7
1.2.1 IAM.....	7
1.2.2 Agencies and Dependencies.....	10
1.2.3 Workspace.....	32
1.3 Configuration Practices in Typical Scenarios.....	32
1.3.1 Assigning Permissions to Individual Users for Using ModelArts.....	33
1.3.2 Separately Assigning Permissions to Administrators and Developers.....	35
1.3.3 Viewing the Notebook Instances of All IAM Users Under One Tenant Account.....	43
1.3.4 Logging In to a Training Container Using Cloud Shell.....	44
1.3.5 Prohibiting a User from Using a Public Resource Pool.....	46
2 Model Development (Custom Algorithms in Training Jobs of the New Version)	49
2.1 Using a Custom Algorithm to Build a Handwritten Digit Recognition Model.....	49
3 Model Inference.....	68
3.1 Creating a Custom Image and Using It to Create an AI Application.....	68
3.2 End-to-End O&M of Inference Services.....	72
3.3 Creating an AI Application Using a Custom Engine.....	75
3.4 Using a Large Model to Create an AI Application and Deploying a Real-Time Service.....	79
3.5 High-Speed Access to Inference Services Through VPC Peering.....	83

1 Permissions Management

1.1 Basic Concepts

ModelArts allows you to configure fine-grained permissions for refined management of resources and permissions. This is commonly used by large enterprises, but it is complex for individual users. It is recommended that individual users configure permissions for using ModelArts by referring to [Assigning Permissions to Individual Users for Using ModelArts](#).

NOTE

If you meet any of the following conditions, read this document.

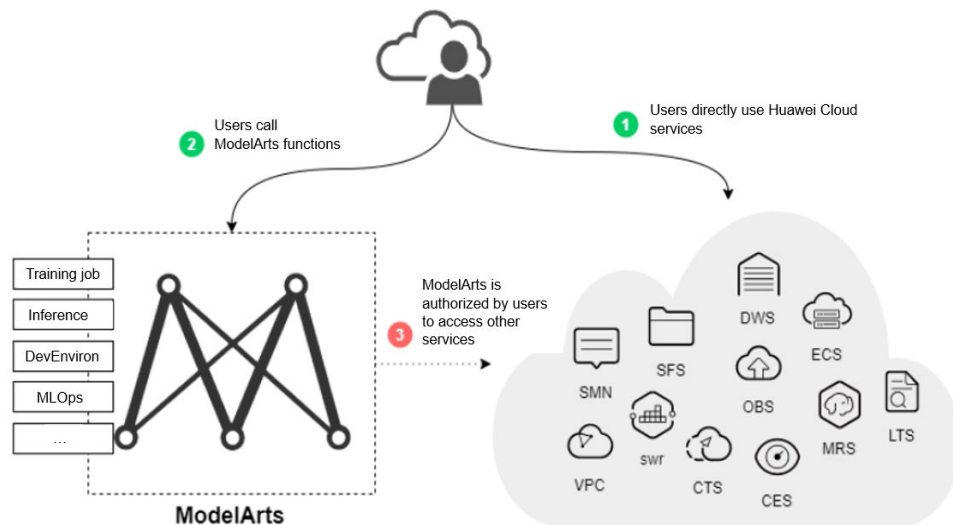
- You are an enterprise user, and
 - There are multiple departments in your enterprise, and you need to control users' permissions so that users in different departments can access only their dedicated resources and functions.
 - There are multiple roles (such as administrators, algorithm developers, and application O&M personnel) in your enterprise. You need them to use only specific functions.
 - There are logically multiple environments (such as the development environment, pre-production environment, and production environment) and are isolated from each other. You need to control users' permissions on different environments.
 - You need to control permissions of specific IAM user or user group.
- You are an individual user, and you have created multiple IAM users. You need to assign different ModelArts permissions to different IAM users.
- You need to understand the concepts and operations of ModelArts permissions management.

ModelArts uses Identity and Access Management (IAM) for most permissions management functions. Before reading below, learn about *Basic Concepts*. This helps you better understand this document.

To implement fine-grained permissions management, ModelArts provides permission control, agency authorization, and workspace. The following describes the details.

ModelArts Permissions and Agencies

Figure 1-1 Permissions management



Exposed ModelArts functions are controlled through IAM permissions. For example, if you as an IAM user need to create a training job on ModelArts, you must have the **modelarts:trainJob:create** permission. For details about how to assign permissions to a user (you need to add the user to a user group and then assign permissions to the user group), see *Permissions Management*.

ModelArts must access other services for AI computing. For example, ModelArts must access OBS to read your data for training. For security purposes, ModelArts must be authorized to access other cloud services. This is agency authorization.

The following summarizes permissions management:

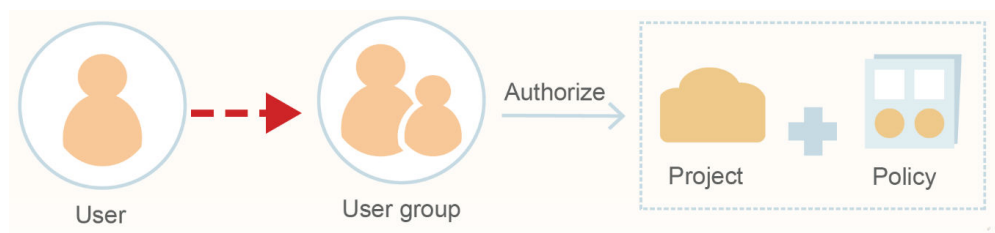
- Your access to any cloud service is controlled through IAM. You must have the permissions of the cloud service. (The required service permissions vary depending on the functions you use.)
- To use ModelArts functions, you need to grant permissions through IAM.
- ModelArts must be authorized by you to access other cloud services for AI computing.

ModelArts Permissions Management

By default, new IAM users do not have any permissions assigned. You need to add a user to one or more groups, and assign permissions policies or roles to these groups. Users inherit permissions of the groups to which they are added. This process is called authorization. After authorization, users can perform operations on ModelArts based on permissions.

CAUTION

ModelArts is a project-level service deployed and accessed in specific physical regions. When you authorize an agency, you can set the scope for the permissions you select to all resources, enterprises projects, or region-specific projects. If you specify region-specific projects, the selected permissions will be applied to resources in these projects.



When assigning permissions to a user group, IAM does not directly assign specific permissions to the user group. Instead, IAM needs to add the permissions to a policy and then assign the policy to the user group. To facilitate user permissions management, each cloud service provides some preset policies for you to directly use. If the preset policies cannot meet your requirements of fine-grained permissions management, you can customize policies.

Table 1-1 lists all the preset system-defined policies supported by ModelArts.

Table 1-1 System-defined policies supported by ModelArts

Policy	Description	Type
ModelArts FullAccess	Administrator permissions for ModelArts. Users granted these permissions can operate and use ModelArts.	System-defined policy
ModelArts CommonOperations	Common user permissions for ModelArts. Users granted these permissions can operate and use ModelArts, but cannot manage dedicated resource pools.	System-defined policy
ModelArts Dependency Access	Permissions on Dependent Services for ModelArts	System-defined policy

Generally, ModelArts FullAccess is assigned only to administrators. If fine-grained management is not required, assigning ModelArts CommonOperations to all users will meet the development requirements of most small teams. If you want to customize policies for fine-grained permissions management, see [IAM](#).

 NOTE

When you assign ModelArts permissions to a user, the system does not automatically assign the permissions of other services to the user. This ensures security and prevents unexpected unauthorized operations. In this case, however, you must separately assign permissions of different services to users so that they can perform some ModelArts operations.

For example, if an IAM user needs to use OBS data for training and the ModelArts training permission has been configured for the IAM user, the IAM user still needs to be assigned with the OBS read, write, and list permissions. The OBS list permission allows you to select the training data path on ModelArts. The read permission is used to preview data and read data for training. The write permission is used to save training results and logs.

- For individual users or small organizations, it is a good practice to configure the **Tenant Administrator** policy that applies to global services for IAM users. In this way, IAM users can obtain all user permissions except IAM. However, this may cause security issues. (For an individual user, its default IAM user belongs to the **admin** user group and has the **Tenant Administrator** permission.)
- If you want to restrict user operations, configure the minimum permissions of OBS for ModelArts users. For details about fine-grained permissions management of other cloud services, see the corresponding cloud service documents.

ModelArts Agency Authorization

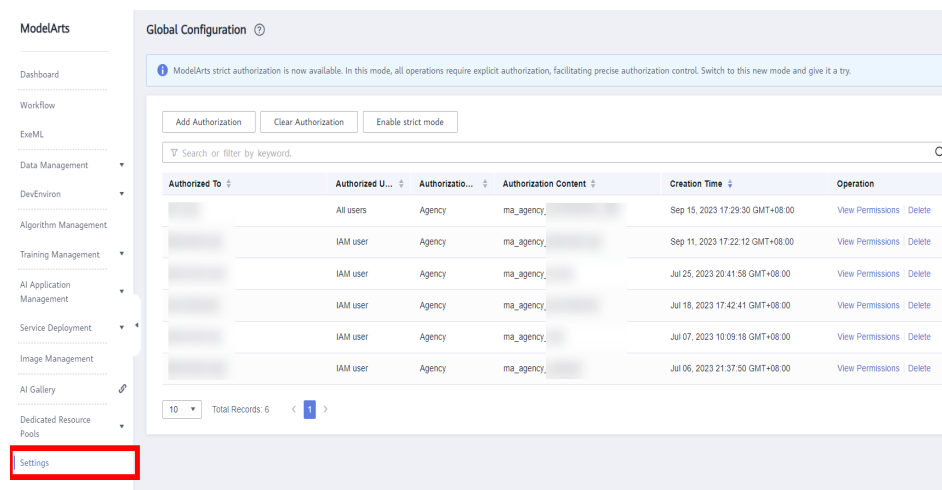
ModelArts must be authorized by users to access other cloud services for AI computing. In the IAM permission system, such authorization is performed through agencies.

To simplify agency authorization, ModelArts supports automatic agency authorization configuration. You only need to configure an agency for yourself or specified users on the **Global Configuration** page of the ModelArts console.

 NOTE

- Only users with the IAM agency management permission can perform this operation. Generally, members in the IAM admin user group have this permission.
- ModelArts agency authorization is region-specific, which means that you must perform agency authorization in each region you use.

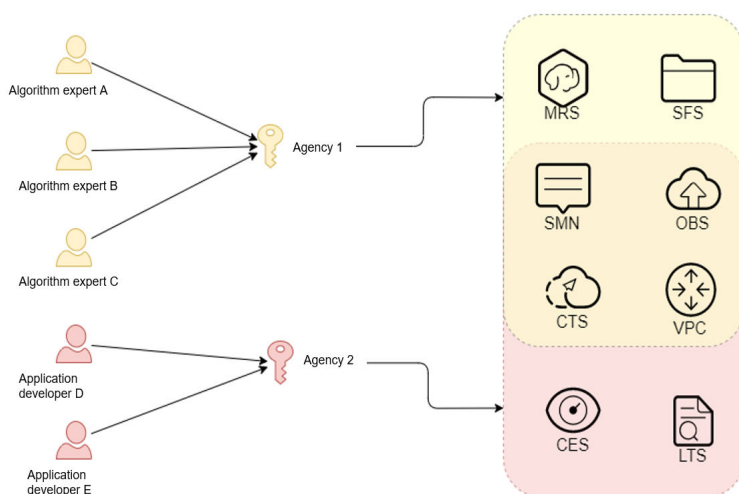
Figure 1-2 Settings



On the **Global Configuration** page of the ModelArts console, after you click **Add Authorization**, you can configure an agency for a specific user or all users. Generally, an agency named **modelarts_agency_<Username>_Random ID** is created by default. In the **Permissions** area, you can select the preset permission configuration or select the required policies. If both options cannot meet your requirements, you can create an agency on the IAM management page (you need to delegate ModelArts to access your resources), and then use an existing agency instead of adding an agency on the **Add Authorization** page.

ModelArts associates multiple users with one agency. This means that if two users need to configure the same agency, you do not need to create an agency for each user. Instead, you only need to configure the same agency for the two users.

Figure 1-3 Mapping between users and agencies



NOTE

Each user can use ModelArts only after being associated with an agency. However, even if the permissions assigned to the agency are insufficient, no error is reported when the API is called. An error occurs only when the system uses unauthorized functions. For example, you enable message notification when creating a training job. Message notification requires SMN authorization. However, an error occurs only when messages need to be sent for the training job. The system ignores some errors, and other errors may cause job failures. When you implement permission minimization, ensure that you will still have sufficient permissions for the required operations on ModelArts.

Strict Authorization

In strict authorization mode, explicit authorization by the account administrator is required for IAM users to access ModelArts. The administrator can add the required ModelArts permissions to common users through authorization policies.

In non-strict authorization mode, IAM users can use ModelArts without explicit authorization. The administrator needs to configure the deny policy for IAM users to prevent them from using some ModelArts functions.

The administrator can change the authorization mode on the **Global Configuration** page.

NOTICE

The strict authorization mode is recommended. In this mode, IAM users must be authorized to use ModelArts functions. In this way, the permission scope of IAM users can be accurately controlled, minimizing permissions granted to IAM users.

Managing Resource Access Using Workspaces

Workspace enables enterprise customers to split their resources into multiple spaces that are logically isolated and to manage access to different spaces. As an enterprise user, you can submit the request for enabling the workspace function to your technical support manager.

After workspace is enabled, a default workspace is created. All resources you have created are in this workspace. A workspace is like a ModelArts twin. You can switch between workspaces in the upper left corner of the ModelArts console. Jobs in different workspaces do not affect each other.

When creating a workspace, you must bind it to an enterprise project. Multiple workspaces can be bound to the same enterprise project, but one workspace cannot be bound to multiple enterprise projects. You can use workspaces for refined restrictions on resource access and permissions of different users. The restrictions are as follows:

- Users must be authorized to access specific workspaces (this must be configured on the pages for creating and managing workspaces). This means that access to AI assets such as datasets and algorithms can be managed using workspaces.
- In the preceding permission authorization operations, if you set the scope to enterprise projects, the authorization takes effect only for workspaces bound to the selected projects.

 NOTE

- Restrictions on workspaces and permission authorization take effect at the same time. That is, a user must have both the permission to access the workspace and the permission to create training jobs (the permission applies to this workspace) so that the user can submit training jobs in this workspace.
- If you have enabled an enterprise project but have not enabled a workspace, all operations are performed in the default enterprise project. Ensure that the permissions on the required operations apply to the default enterprise project.
- The preceding restrictions do not apply to users who have not enabled any enterprise project.

Summary

Key features of ModelArts permissions management:

- If you are an individual user, you do not need to consider fine-grained permissions management. Your account has all permissions to use ModelArts by default.
- All functions of ModelArts are controlled by IAM. You can use IAM authorization to implement fine-grained permissions management for specific users.

- All users (including individual users) can use specific functions only after agency authorization on ModelArts (**Settings > Add Authorization**). Otherwise, unexpected errors may occur.
- If you have enabled the enterprise project function, you can also enable ModelArts workspace and use both basic authorization and workspace for refined permissions management.

1.2 Permission Management Mechanisms

1.2.1 IAM

This section describes the IAM permission configurations for all ModelArts functions.

IAM Permissions

If no fine-grained authorization policy is configured for a user created by the administrator, the user has all permissions of ModelArts by default. To control user permissions, the administrator needs to add the user to a user group on IAM and configure fine-grained authorization policies for the user group. In this way, the user obtains the permissions defined in the policies before performing operations on cloud service resources.

You can grant users permissions by using roles and policies.

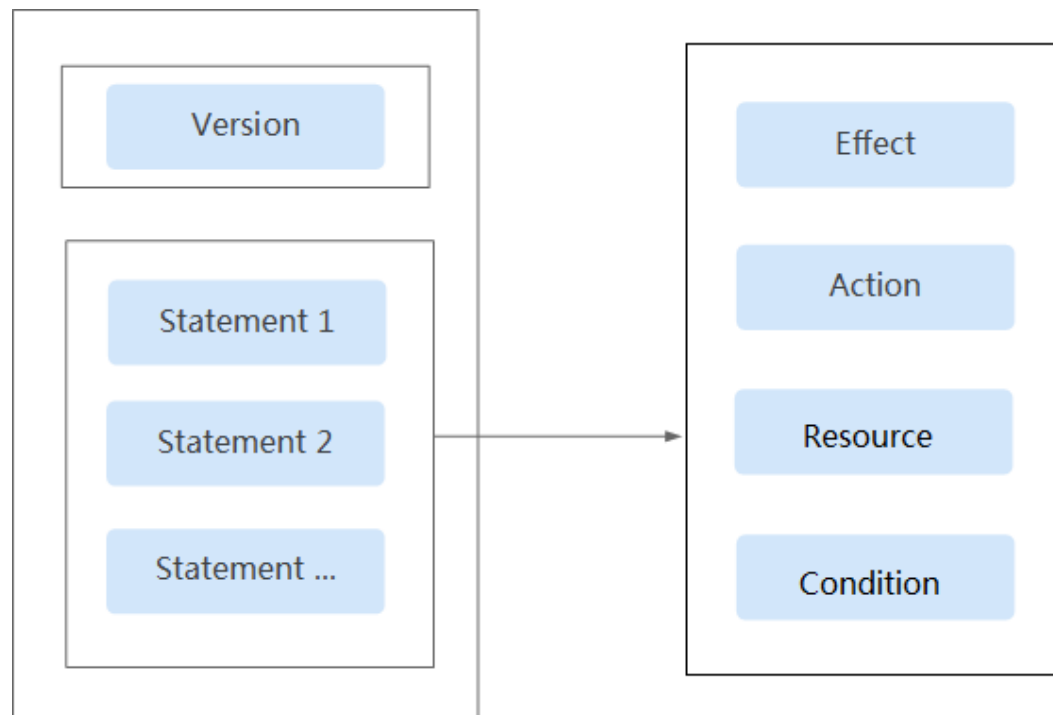
- Roles are a type of coarse-grained authorization mechanism that defines permissions related to user responsibilities. Only a limited number of service-level roles are available. When using roles to grant permissions, you must also assign other roles on which the permissions depend to take effect. Roles are not ideal for fine-grained authorization and secure access control.
- Policies are a type of fine-grained authorization mechanism that defines permissions required to perform operations on specific cloud resources under certain conditions. This type of authorization is more flexible and ideal for secure access control. For example, you can grant ECS users permissions that only allow them to manage a certain type of ECS.

ModelArts does not support role-based authorization. It supports only policy-based authorization.

Policy Structure

A policy consists of a version and one or more statements (indicating different actions).

Figure 1-4 Policy structure



Policy Parameters

The following describes policy parameters. You can create custom policies by specifying the parameters.

Table 1-2 Policy parameters

Parameter		Description	Value
Version		Policy version	1.1 : indicates policy-based access control.
Statement: authorization statement of a policy	Effect	Whether to allow or deny the operations defined in the action	<ul style="list-style-type: none"> • Allow: indicates the operation is allowed. • Deny: indicates the operation is not allowed. <p>NOTE If the policy used to grant user permissions contains both Allow and Deny for the same action, Deny takes precedence.</p>

Parameter		Description	Value
	Action	Operation to be performed on the service	Format: " <i>Service name.Resource type.Action</i> ". Wildcard characters (*) are supported, indicating all options. Example: modelarts:notebook:list : indicates the permission to view a notebook instance list. modelarts indicates the service name, notebook indicates the resource type, and list indicates the operation. View all actions of a service in its <i>API Reference</i> .
	Condition	Condition for a policy to take effect, including condition keys and operators	Format: " <i>Condition operator:{Condition key:[Value 1,Value 2]}</i> " If you set multiple conditions, the policy takes effect only when all the conditions are met. Example: StringEndWithIfExists": {"g:UserName":["specialCharacter"]} : The statement is valid for users whose names end with specialCharacter .
	Resource	Resources on which a policy takes effect	Format: <i>Service name.Region.Account ID.Resource type.Resource path</i> . Wildcard characters (*) are supported, indicating all resources. NOTE ModelArts authorization does not allow you to specify a resource path.

ModelArts Resource Types

During policy-based authorization, the administrator can select the authorization scope based on ModelArts resource types. The following table lists the resource types supported by ModelArts:

Table 1-3 ModelArts resource types

Resource Type	Description
notebook	Notebook instances in DevEnviron
exemlProject	ExeML projects
exemlProjectInf	ExeML-powered real-time inference service

Resource Type	Description
exemlProjectTrain	ExeML-powered training jobs
exemlProjectVersion	ExeML project version
workflow	Workflow
pool	Dedicated resource pool
network	Networking of a dedicated resource pool
trainJob	Training job
trainJobLog	Runtime logs of a training job
trainJobInnerModel	Preset model
trainJobVersion	Version of a training job (supported by old-version training jobs that will be discontinued soon)
trainConfig	Configuration of a training job (supported by old-version training jobs that will be discontinued soon)
tensorboard	Visualization job of training results (supported by old-version training jobs that will be discontinued soon)
model	Models
service	Real-time service
nodeservice	Edge service
workspace	Workspace
dataset	Dataset
dataAnnotation	Dataset labels
aiAlgorithm	Algorithm for training jobs
image	Image

ModelArts Resource Permissions

For details, see "Permissions Policies and Supported Actions" in *ModelArts API Reference*.

1.2.2 Agencies and Dependencies

Function Dependency

Function Dependency Policies

When using ModelArts to develop algorithms or manage training jobs, you are required to use other Cloud services. For example, before submitting a training job, select an OBS path for storing the dataset and logs, respectively. Therefore, when configuring fine-grained authorization policies for a user, the administrator must configure dependent permissions so that the user can use required functions.

 **NOTE**

If you use ModelArts as the root user (default IAM user with the same name as the account), the root user has all permissions by default.

Table 1-4 Basic configuration

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Global configuration	IAM	iam:users:listUsers	Obtain a user list. This action is required by the administrator only.
Basic function	IAM	iam:tokens:assume	(Mandatory) Use an agency to obtain temporary authentication credentials.

Table 1-5 Managing workspaces

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Workspace	IAM	iam:users:listUsers	Authorize an IAM user to use a workspace.
	ModelArts	modelarts:*:*delete*	Clear resources in a workspace when deleting it.

Table 1-6 Managing notebook instances

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Lifecycle management of development environment instances	ModelArts	modelarts:notebook:create modelarts:notebook:list modelarts:notebook:get modelarts:notebook:update modelarts:notebook:delete modelarts:notebook:start modelarts:notebook:stop modelarts:notebook:updateStopPolicy modelarts:image:delete modelarts:image:list modelarts:image:create modelarts:image:get modelarts:pool:list modelarts:tag:list modelarts:network:get aom:metric:get aom:metric:list aom:alarm:list	Start, stop, create, delete, and update an instance.
Dynamically mounting storage	ModelArts	modelarts:notebook:listMountedStorages modelarts:notebook:mountStorage modelarts:notebook:getMountedStorage modelarts:notebook:umountStorage	Dynamically mount storage.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
	OBS	obs:bucket:ListAllMyBuckets obs:bucket:ListBucket	
Image management	ModelArts	modelarts:image:register modelarts:image:listGroup	Register and view an image on the Image Management page.
Saving an image	SWR	SWR Admin	The SWR Admin policy contains the maximum scope of SWR permissions, which can be used to: <ul style="list-style-type: none"> • Save a running development environment instance as an image. • Create a notebook instance using a custom image.
Using the SSH function	ECS	ecs:serverKeyPairs:list ecs:serverKeyPairs:get ecs:serverKeyPairs:delete ecs:serverKeyPairs:create	Configure a login key for a notebook instance.
Mounting an SFS Turbo file system	SFS Turbo	SFS Turbo FullAccess	Read and write an SFS directory as an IAM user. Mount an SFS file system that is not created by you to a notebook instance using a dedicated resource pool.
Viewing all Instances	ModelArts	modelarts:notebook:listAllNotebooks	View development environment instances of all users on the ModelArts management console. This action is required by the development environment instance administrator.
	IAM	iam:users:listUsers	

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Local VS Code plug-in or PyCharm Toolkit	ModelArts	modelarts:notebook:listAllNotebooks modelarts:trainJob:create modelarts:trainJob:list modelarts:trainJob:update modelarts:trainJobVersion:delete modelarts:trainJob:get modelarts:trainJob:logExport modelarts:workspace:getQuotas (This policy is required if the workspace function is enabled.)	Access a notebook instance from local VS Code and submit training jobs.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
	OBS	obs:bucket:ListAllMybuckets obs:bucket:HeadBucket obs:bucket:ListBucket obs:bucket:GetBucketLocation obs:object:GetObject obs:object:GetObjectVersion obs:object:PutObject obs:object>DeleteObject obs:object>DeleteObjectVersion obs:object:ListMultipartUploadParts obs:object:AbortMultipartUpload obs:object:GetObjectAcl obs:object:GetObjectVersionAcl obs:bucket:PutBucketAcl obs:object:PutObjectAcl obs:object:ModifyObjectMetadata	
	IAM	iam:projects:listProjects	Obtain an IAM project list through local PyCharm for access configurations.

Table 1-7 Managing training jobs

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Training management	ModelArts	modelarts:trainJob:* modelarts:trainJobLog:* modelarts:aiAlgorithm:* modelarts:image:list	Create a training job and view training logs.
		modelarts:workspace:getQuotas	Obtain a workspace quota. This policy is required if the workspace function is enabled.
		modelarts:tag:list	Use Tag Management Service (TMS) in a training job.
	IAM	iam:credentials:listCredentials iam:agencies:listAgencies	Use the configured agency authorization.
	SFS Turbo	sfsturbo:shares:getShare sfsturbo:shares:getAllShares	Use SFS Turbo in a training job.
	SWR	swr:repository:listTags swr:repository:getRepository swr:repository:listRepositories	Use a custom image to create a training job.
	SMN	smn:topic:publish smn:topic:list	Notify training job status changes through SMN.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
	OBS	obs:bucket:ListAllMybuckets obs:bucket:HeadBucket obs:bucket:ListBucket obs:bucket:GetBucketLocation obs:object:GetObject obs:object:GetObjectVersion obs:object:PutObject obs:object:DeleteObject obs:object:DeleteObjectVersion obs:object:ListMultipartUploadParts obs:object:AbortMultipartUpload obs:object:GetObjectAcl obs:object:GetObjectVersionAcl obs:bucket:PutBucketAcl obs:object:PutObjectAcl obs:object:ModifyObjectMetadata	Run a training job using a dataset in an OBS bucket.

Table 1-8 Using workflows

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Using a dataset	ModelArts	modelarts:dataset:getDataset modelarts:dataset:createDataset modelarts:dataset:createDatasetVersion modelarts:dataset:createImportTask modelarts:dataset:updateDataset modelarts:processTask:createProcessTask modelarts:processTask:getProcessTask modelarts:dataset:listDatasets	Use ModelArts datasets in a workflow.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Managing AI applications	ModelArts	modelarts:model:list modelarts:model:get modelarts:model:create modelarts:model:delete modelarts:model:update	Manage ModelArts AI applications in a workflow.
Deploying a service	ModelArts	modelarts:service:get modelarts:service:create modelarts:service:update modelarts:service:delete modelarts:service:getLogs	Manage ModelArts real-time services in a workflow.
Training jobs	ModelArts	modelarts:trainJob:get modelarts:trainJob:create modelarts:trainJob:list modelarts:trainJobVersion:list modelarts:trainJobVersion:create modelarts:trainJob:delete modelarts:trainJobVersion:delete modelarts:trainJobVersion:stop	Manage ModelArts training jobs in a workflow.
Workspace	ModelArts	modelarts:workspace:get modelarts:workspace:getQuotas	Use ModelArts workspaces in a workflow.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Managing data	OBS	obs:bucket:ListAllMybuckets (Obtaining a bucket list) obs:bucket:HeadBucket (Obtaining bucket metadata) obs:bucket:ListBucket (Listing objects in a bucket) obs:bucket:GetBucketLocation (Obtaining the bucket location) obs:object:GetObject (Obtaining object content and metadata) obs:object:GetObjectVersion (Obtaining object content and metadata) obs:object:PutObject (Uploading objects using PUT method, uploading objects using POST method, copying objects, appending an object, initializing a multipart task, uploading parts, and merging parts) obs:object>DeleteObject (Deleting an object or batch deleting objects) obs:object>DeleteObjectVersion (Deleting an object or batch deleting objects) obs:object:ListMultipartUploadParts (Listing uploaded parts) obs:object:AbortMultipartUpload (Aborting multipart uploads) obs:object:GetObjectAcl (Obtaining an object ACL) obs:object:GetObjectVersionAcl (Obtaining an object ACL) obs:bucket:PutBucketAcl (Configuring a bucket ACL) obs:object:PutObjectAcl (Configuring an object ACL)	Use OBS data in a workflow.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Executing a workflow	IAM	iam:users:listUsers (Obtaining users) iam:agencies:getAgency (Obtaining details about a specified agency) iam:tokens:assume (Obtaining an agency token)	Call other ModelArts services when the workflow is running.
Integrating DLI	DLI	dli:jobs:get (Obtaining job details) dli:jobs:list_all (Viewing a job list) dli:jobs:create (Creating a job)	Integrate DLI into a workflow.
Integrating MRS	MRS	mrs:job:get (Obtaining job details) mrs:job:submit (Creating and executing a job) mrs:job:list (Viewing a job list) mrs:job:stop (Stopping a job) mrs:job:batchDelete (Batch deleting jobs) mrs:file:list (Viewing a file list)	Integrate MRS into a workflow.

Table 1-9 Managing AI applications

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Managing AI applications	SWR	swr:repository:deleteRepository swr:repository:deleteTag swr:repository:getRepository swr:repository:listTags	Import a model from a custom image. Use a custom engine when importing a model from OBS.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
	OBS	obs:bucket:ListAllMybuckets (Obtaining a bucket list) obs:bucket:HeadBucket (Obtaining bucket metadata) obs:bucket:ListBucket (Listing objects in a bucket) obs:bucket:GetBucketLocation (Obtaining the bucket location) obs:object:GetObject (Obtaining object content and metadata) obs:object:GetObjectVersion (Obtaining object content and metadata) obs:object:PutObject (Uploading objects using PUT method, uploading objects using POST method, copying objects, appending an object, initializing a multipart task, uploading parts, and merging parts) obs:object>DeleteObject (Deleting an object or batch deleting objects) obs:object>DeleteObjectVersion (Deleting an object or batch deleting objects) obs:object:ListMultipartUploadParts (Listing uploaded parts) obs:object:AbortMultipartUpload (Aborting multipart uploads) obs:object:GetObjectAcl (Obtaining an object ACL) obs:object:GetObjectVersionAcl (Obtaining an object ACL) obs:bucket:PutBucketAcl (Configuring a bucket ACL) obs:object:PutObjectAcl (Configuring an object ACL)	Import a model from a template. Specify an OBS path for model conversion.

Table 1-10 Managing service deployment

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Real-time services	LTS	lts:logs:list (Obtaining the log list)	Show LTS logs.
	OBS	obs:bucket:GetBucketPolicy (Obtaining a bucket policy) obs:bucket:HeadBucket (Obtaining bucket metadata) obs:bucket:ListAllMyBuckets (Obtaining a bucket list) obs:bucket:PutBucketPolicy (Configuring a bucket policy) obs:bucket>DeleteBucketPolicy (Deleting a bucket policy)	Mount external volumes to a container when services are running.
Batch services	OBS	obs:object:GetObject (Obtaining object content and metadata) obs:object:PutObject (Uploading objects using PUT method, uploading objects using POST method, copying objects, appending an object, initializing a multipart task, uploading parts, and merging parts) obs:bucket>CreateBucket (Creating a bucket) obs:bucket:ListBucket (Listing objects in a bucket) obs:bucket:ListAllMyBuckets (Obtaining a bucket list)	Create batch services and perform batch inference.
Edge services	CES	ces:metricData:list: (Obtaining metric data)	View monitoring metrics.
	IEF	ief:deployment:delete (Deleting a deployment)	Manage edge services.

Table 1-11 Managing datasets

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Managing datasets and labels	OBS	obs:bucket:ListBucket (Listing objects in a bucket) obs:object:GetObject (Obtaining object content and metadata) obs:object:PutObject (Uploading objects using PUT method, uploading objects using POST method, copying objects, appending an object, initializing a multipart task, uploading parts, and merging parts) obs:object:DeleteObject (Deleting an object or batch deleting objects) obs:bucket:HeadBucket (Obtaining bucket metadata) obs:bucket:GetBucketAcl (Obtaining a bucket ACL) obs:bucket:PutBucketAcl (Configuring a bucket ACL) obs:bucket:GetBucketPolicy (Obtaining a bucket policy) obs:bucket:PutBucketPolicy (Configuring a bucket policy) obs:bucket:DeleteBucketPolicy (Deleting a bucket policy) obs:bucket:PutBucketCORS (Configuring or deleting CORS rules of a bucket) obs:bucket:GetBucketCORS (Obtaining the CORS rules of a bucket) obs:object:PutObjectAcl (Configuring an object ACL)	Manage datasets in OBS. Label OBS data. Create a data management job.
Managing table datasets	DLI	dli:database:displayAllDatabases dli:database:displayAllTables dli:table:describe_table	Manage DLI data in a dataset.
Managing table datasets	DWS	dws:openAPICluster:list dws:openAPICluster:getDetail	Manage DWS data in a dataset.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Managing table datasets	MRS	mrs:job:submit mrs:job:list mrs:cluster:list mrs:cluster:get	Manage MRS data in a dataset.
Auto labeling	ModelArts	modelarts:service:list modelarts:model:list modelarts:model:get modelarts:model:create modelarts:trainJobInnerModel:list modelarts:workspace:get modelarts:workspace:list	Enable auto labeling.
Team labeling	IAM	iam:projects:listProjects (Obtaining tenant projects) iam:users:listUsers (Obtaining users) iam:agencies:createAgency (Creating an agency) iam:quotas:listQuotasForProject (Obtaining the quotas of a project)	Manage labeling teams.

Table 1-12 Managing resources

Application Scenario	Dependent Service	Dependent Policy	Supported Function
Managing resource pools	BSS	bss:coupon:view bss:order:view bss:balance:view bss:discount:view bss:renewal:view bss:bill:view bss:contract:update bss:order:pay bss:unsubscribe:update bss:renewal:update bss:order:update	Create, renew, and unsubscribe from a resource pool. Dependent permissions must be configured in the IAM project view.

Application Scenario	Dependent Service	Dependent Policy	Supported Function
	ECS	ecs:availabilityZones:list	Show AZs. Dependent permissions must be configured in the IAM project view.
Network management	VPC	vpc:routes:create vpc:routes:list vpc:routes:get vpc:routes:delete vpc:peerings:create vpc:peerings:accept vpc:peerings:get vpc:peerings:delete vpc:routeTables:update vpc:routeTables:get vpc:routeTables:list vpc:vpcs:create vpc:vpcs:list vpc:vpcs:get vpc:vpcs:delete vpc:subnets:create vpc:subnets:get vpc:subnets:delete vpcep:endpoints:list vpcep:endpoints:create vpcep:endpoints:delete vpcep:endpoints:get vpc:ports:create vpc:ports:get vpc:ports:update vpc:ports:delete vpc:networks:create vpc:networks:get vpc:networks:update vpc:networks:delete	Create and delete ModelArts networks, and interconnect VPCs. Dependent permissions must be configured in the IAM project view.

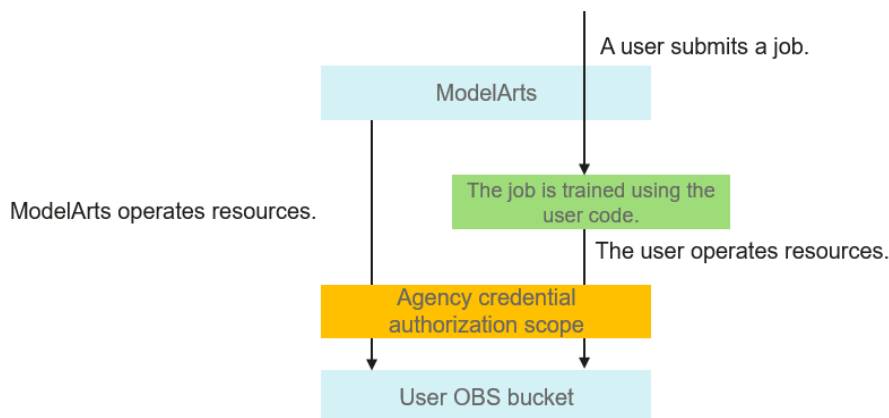
Application Scenario	Dependent Service	Dependent Policy	Supported Function
	SFS Turbo	sfsturbo:shares:addShareNic sfsturbo:shares:deleteShareNic sfsturbo:shares:showShareNic sfsturbo:shares:listShareNics	Interconnect your network with SFS Turbo. Dependent permissions must be configured in the IAM project view.
Edge resource pool	IEF	ief:node:list ief:group:get ief:application:list ief:application:get ief:node:listNodeCert ief:node:get ief:IEFInstance:get ief:deployment:list ief:group:listGroupInstanceState ief:IEFInstance:list ief:deployment:get ief:group:list	Add, delete, modify, and search for edge pools

Agency authorization

To simplify operations when you use ModelArts to run jobs, certain operations are automatically performed on the ModelArts backend, for example, downloading the datasets in an OBS bucket to a workspace before a training job is started and dumping training job logs to the OBS bucket.

ModelArts does not save your token authentication credentials. Before performing operations on your resources (such as OBS buckets) in a backend asynchronous job, you are required to explicitly authorize ModelArts through an IAM agency. ModelArts will use the agency to obtain a temporary authentication credential for performing operations on your resources. For details, see [Adding Authorization](#).

Figure 1-5 Agency authorization



As shown in [Figure 1-5](#), after authorization is configured on ModelArts, ModelArts uses the temporary credential to access and operate your resources, relieving you from some complex and time-consuming operations. The agency credential will also be synchronized to your jobs (including notebook instances and training jobs). You can use the agency credential to access your resources in the jobs.

You can use either of the following methods to authorize ModelArts using an agency:

One-click authorization

ModelArts provides one-click automatic authorization. You can quickly configure agency authorization on the **Global Configuration** page of ModelArts. Then, ModelArts will automatically create an agency for you and configure it in ModelArts.

In this mode, the authorization scope is specified based on the preset system policies of dependent services to ensure sufficient permissions for using services. The created agency has almost all permissions of dependent services. If you want to precisely control the scope of permissions granted to an agency, use the second method.

Custom authorization

The administrator creates different agency authorization policies for different users in IAM, and configures the created agency for ModelArts users. When creating an agency for an IAM user, the administrator specifies the minimum permissions for the agency based on the user's permissions to control the resources that the user can access when they use ModelArts.

Risks in Unauthorized Operations

The agency authorization of a user is independent. Theoretically, the agency authorization scope of a user can be beyond the authorization scope of the authorization policy configured for the user group. Any improper configuration will result in unauthorized operations.

To prevent unauthorized operations, only a tenant administrator is allowed to configure agencies for users in the ModelArts global configuration to ensure the security of agency authorization.

Minimal Agency Authorization

When configuring agency authorization, an administrator must strictly control the authorization scope.

ModelArts asynchronously and automatically performs operations such as job preparation and clearing. The required agency authorization is within the basic authorization scope. If you use only some functions of ModelArts, the administrator can filter out the basic permissions that are not used according to the agency authorization configuration. Conversely, if you need to obtain resource permissions beyond the basic authorization scope in a job, the administrator can add new permissions to the agency authorization configuration. In a word, the agency authorization scope must be minimized and customized based on service requirements.

Basic Agency Authorization Scope

To customize the permissions for an agency, select permissions based on your service requirements.

Table 1-13 Basic agency authorization for a development environment

Application Scenario	Dependent Service	Agency Authorization	Description	Configuration Suggestion
JupyterLab	OBS	obs:object:DeleteObject obs:object:GetObject obs:object:GetObjectVersion obs:bucket>CreateBucket obs:bucket:ListBucket obs:bucket:ListAllMyBuckets obs:object:PutObject obs:bucket:GetBucketAcl obs:bucket:PutBucketAcl obs:bucket:PutBucketCORS	Use OBS to upload and download data in JupyterLab through ModelArts notebook.	Recommended
Development environment monitoring	AOM	aom:alarm:put	Call the AOM API to obtain monitoring data and events of notebook instances and display them in ModelArts notebook.	Recommended

Table 1-14 Basic agency authorization for training jobs

Application Scenario	Dependent Service	Agency Authorization	Description
Training jobs	OBS	obs:bucket:ListBucket obs:object:GetObject obs:object:PutObject	Download data, models, and code before starting a training job. Upload logs and models when a training job is running.

Table 1-15 Basic agency authorization for deploying services

Application Scenario	Dependent Service	Agency Authorization	Description
Real-time services	LTS	lts:groups:create lts:groups:list lts:topics:create lts:topics:delete lts:topics:list	Configure LTS for reporting logs of real-time services.
Batch services	OBS	obs:bucket:ListBucket obs:object:GetObject obs:object:PutObject	Run a batch service.
Edge services	IEF	ief:deployment:list ief:deployment:create ief:deployment:update ief:deployment:delete ief:node:createNodeCert ief:iefInstance:list ief:node:list	Deploy an edge service using IEF.

Table 1-16 Basic agency authorization for managing data

Application Scenario	Dependent Service	Agency Authorization	Description
Dataset and data labeling	OBS	obs:object:GetObject obs:object:PutObject obs:object>DeleteObject obs:object:PutObjectAcl obs:bucket:ListBucket obs:bucket:HeadBucket obs:bucket:GetBucketAcl obs:bucket:PutBucketAcl obs:bucket:GetBucketPolicy obs:bucket:PutBucketPolicy obs:bucket>DeleteBucketPolicy obs:bucket:PutBucketCORS obs:bucket:GetBucketCORS	Manage datasets in an OBS bucket.
Labeling data	ModelArts inference	modelarts:service:get modelarts:service:create modelarts:service:update	Perform auto labeling based on ModelArts inference.

Table 1-17 Basic agency authorization for managing dedicated resource pools

Application Scenario	Dependent Service	Agency Authorization	Description
Network management (New version)	VPC	vpc:routes:create vpc:routes:list vpc:routes:get vpc:routes:delete vpc:peerings:create vpc:peerings:accept vpc:peerings:get vpc:peerings:delete vpc:routeTables:update vpc:routeTables:get vpc:routeTables:list vpc:vpcs:create vpc:vpcs:list vpc:vpcs:get vpc:vpcs:delete vpc:subnets:create vpc:subnets:get vpc:subnets:delete vpcep:endpoints:list vpcep:endpoints:create vpcep:endpoints:delete vpcep:endpoints:get vpc:ports:create vpc:ports:get vpc:ports:update vpc:ports:delete vpc:networks:create vpc:networks:get vpc:networks:update vpc:networks:delete	Create and delete ModelArts networks, and interconnect VPCs. Dependent permissions must be configured in the IAM project view.
	SFS Turbo	sfsturbo:shares:addShareNic sfsturbo:shares:deleteShareNic sfsturbo:shares:showShareNic sfsturbo:shares:listShareNics	Interconnect your network with SFS Turbo. Dependent permissions must be configured in the IAM project view.

Application Scenario	Dependent Service	Agency Authorization	Description
Managing resource pools	ECS	ecs:availabilityZones:list	Show AZs. Dependent permissions must be configured in the IAM project view.

1.2.3 Workspace

ModelArts allows you to create multiple workspaces to develop algorithms and manage and deploy models for different service objectives. In this way, the development outputs of different applications are allocated to different workspaces for simplified management.

Workspace supports the following types of access control:

- **PUBLIC:** publicly accessible to tenants (including both tenant accounts and all their user accounts)
- **PRIVATE:** accessible only to the creator and tenant accounts
- **INTERNAL:** accessible to the creator, tenant accounts, and specified IAM user accounts. When **Authorization Type** is set to **INTERNAL**, specify one or more accessible IAM user accounts.

A default workspace is allocated to each IAM project of each account. The access control of the default workspace is **PUBLIC**.

Workspace access control allows the access of only certain users. This function can be used in the following scenarios:

- **Education:** A teacher allocates an **INTERNAL** workspace to each student and allows the workspaces to be accessed only by specified students. In this way, students can separately perform experiments on ModelArts.
- **Enterprises:** An administrator creates a workspace for production tasks and allows only O&M personnel to use the workspace, and creates a workspace for routine debugging and allows only developers to use the workspace. In this way, different enterprise roles can use resources only in a specified workspace.

As an enterprise user, you can submit the request for enabling the workspace function to your technical support.

1.3 Configuration Practices in Typical Scenarios

1.3.1 Assigning Permissions to Individual Users for Using ModelArts

Certain ModelArts functions require access to Object Storage Service (OBS), Software Repository for Container (SWR), and Intelligent EdgeFabric (IEF). Before using ModelArts, your account must be authorized to access these services. Otherwise, these functions will be unavailable.

Constraints

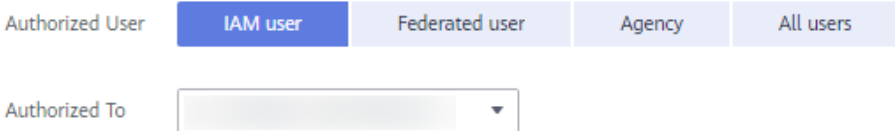
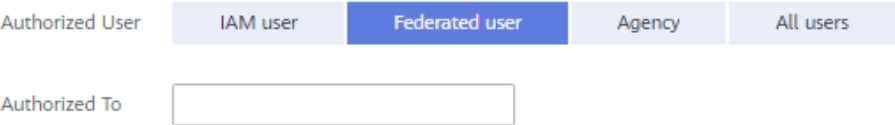
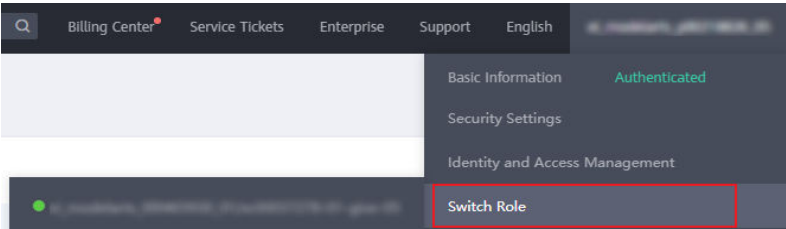
- Only a tenant account can perform agency authorization to authorize the current account or all IAM users under the current account.
- Multiple IAM users or accounts can use the same agency.
- A maximum of 50 agencies can be created under an account.
- If you use ModelArts for the first time, add an agency. Generally, common user permissions are sufficient for your requirements. You can configure permissions for refined permissions management.
- If you have not been authorized, ModelArts will display a message indicating that you have not been authorized when you access the **Add Authorization** page. In this case, contact your administrator to add authorization.

Adding Authorization

1. Log in to the ModelArts management console. In the left navigation pane, choose **Settings**. The **Global Configuration** page is displayed.
2. Click **Add Authorization**. On the **Add Authorization** page that is displayed, configure the parameters.

Table 1-18 Parameters

Parameter	Description
Authorized User	<p>Options: IAM user, Federated user, Agency, and All users</p> <ul style="list-style-type: none"> • IAM user: You can use a tenant account to create IAM users and assign permissions for specific resources. Each IAM user has their own identity credentials (password and access keys) and uses cloud resources based on assigned permissions. • Federated user: A federated user is also called a virtual enterprise user. • Agency: You can create agencies in IAM. • All users: If you select this option, the agency permissions will be granted to all IAM users under the current account, including those created in the future. For individual users, choose All users.

Parameter	Description
Authorized To	<p>This parameter is not displayed when Authorized User is set to All users.</p> <ul style="list-style-type: none"> IAM user: Select an IAM user and configure an agency for the IAM user. Figure 1-6 Selecting an IAM user  Federated user: Enter the username or user ID of the target federated user. Figure 1-7 Selecting a federated user  Agency: Select an agency name. You can use account A to create an agency and configure the agency for account B. When using account B, you can switch the role in the upper right corner of the console to account A and use the agency permissions of account A. Figure 1-8 Switch Role 
Agency	<ul style="list-style-type: none"> Use existing: If there are agencies in the list, select an available one to authorize the selected user. Click the drop-down arrow next to an agency name to view its permission details. Add agency: If there is no available agency, create one. If you use ModelArts for the first time, select Add agency.
Add agency > Agency Name	The system automatically creates a changeable agency name.
Add agency > Permissions > Common User	<p>Common User provides the permissions to use all basic ModelArts functions. For example, you can access data, and create and manage training jobs. Select this option generally.</p> <p>Click View permissions to view common user permissions.</p>

Parameter	Description
Add agency > Permissions > Custom	If you need refined permissions management, select Custom to flexibly assign permissions to the created agency. You can select permissions from the permission list as required.

3. Click **Create**.

Viewing Authorized Permissions

You can view the configured authorizations on the **Global Configuration** page. Click **View Permissions** in the **Authorization Content** column to view the permission details.

Figure 1-9 View Permissions

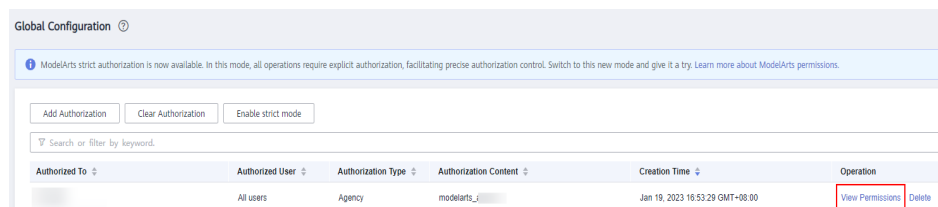


Figure 1-10 View Permissions

View Permissions

Name	Type	Description
DLI FullAccess	System-defined policy	Full permissions for Data Lake Insight.
VPC Administrator	System-defined role	VPC Administrator
EPS FullAccess	System-defined policy	All operations on the Enterprise Project Management service.
CTS Administrator	System-defined role	CTS Administrator
ModelArts CommonOperations	System-defined policy	Common permissions of ModelArts service, except create, update, del...
SFS ReadOnlyAccess	System-defined policy	The read-only permissions to all SFS resources.
OBS Administrator	System-defined policy	Object Storage Service Administrator
DWS Administrator	System-defined role	Data Warehouse Service Administrator
LTS FullAccess	System-defined policy	All permissions of Log Tank service.
CES ReadOnlyAccess	System-defined policy	Read-only permissions for Cloud Eye.

10 Total Records: 12 < 1 2 >

1.3.2 Separately Assigning Permissions to Administrators and Developers

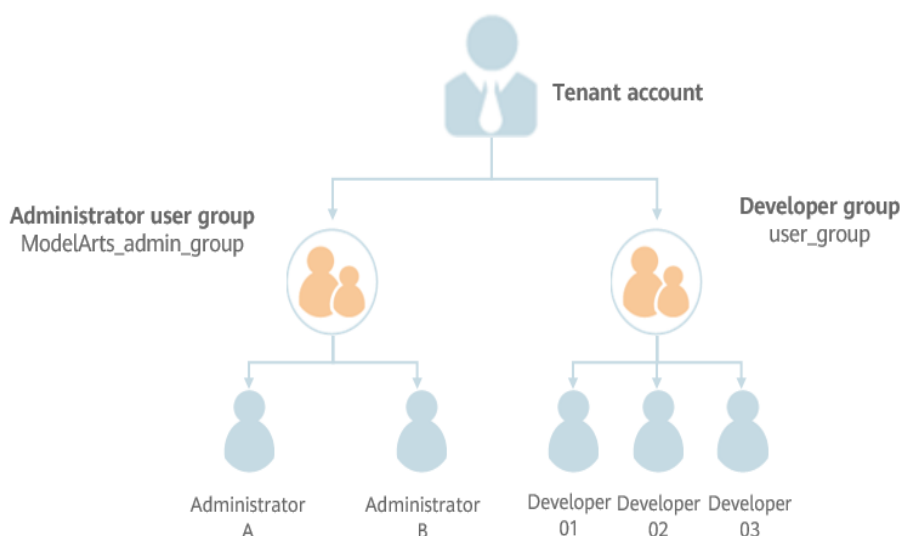
In small- and medium-sized teams, administrators need to globally control ModelArts resources, and developers only need to focus on their own instances. By default, a developer account does not have the **te_admin** permission. The tenant account must configure the required permissions. This section uses notebook as an example to describe how to assign different permissions to administrators and developers through custom policies.

Scenarios

To develop a project using notebook, administrators need full control permissions for using ModelArts dedicated resource pools, and access and operation permissions on all notebook instances.

To use development environments, developers only need operation permissions for using their own instances and dependent services. They do not need to perform operations on ModelArts dedicated resource pools or view notebook instances of other users.

Figure 1-11 Account relationships

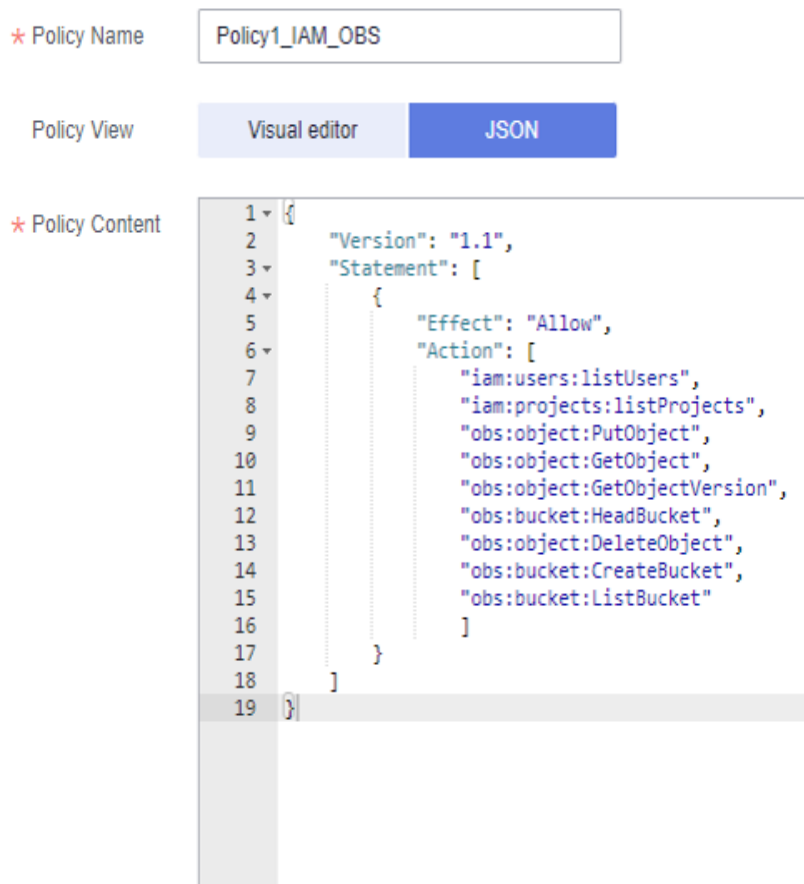


Configuring Permissions for an Administrator

Assign full control permissions to administrators for using ModelArts dedicated resource pools and all notebook instances. The procedure is as follows:

- Step 1** Use a tenant account to create an administrator user group **ModelArts_admin_group** and add administrator accounts to **ModelArts_admin_group**.
- Step 2** Create a custom policy.
1. Log in to the management console using an administrator account, hover over your username in the upper right corner, and click **Identity and Access Management** from the drop-down list to switch to the IAM management console.
 2. Create custom policy 1 and assign IAM and OBS permissions to the user. In the navigation pane of the IAM console, choose **Permissions > Policies/Roles**. Click **Create Custom Policy** in the upper right corner. On the displayed page, enter **Policy1_IAM_OBS** for **Policy Name**, select **JSON** for **Policy View**, configure the policy content, and click **OK**.

Figure 1-12 Custom policy 1



The custom policy **Policy1_IAM_OBS** is as follows, which grants IAM and OBS operation permissions to the user. You can directly copy and paste the content.

```

{
  "Version": "1.1",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iam:users:listUsers",
        "iam:projects:listProjects",
        "obs:object:PutObject",
        "obs:object:GetObject",
        "obs:object:GetObjectVersion",
        "obs:bucket:HeadBucket",
        "obs:object:DeleteObject",
        "obs:bucket:CreateBucket",
        "obs:bucket:ListBucket"
      ]
    }
  ]
}
```

- Repeat **Step 2.2** to create custom policy 2 and grant the user the permissions to perform operations on dependent services ECS, SWR, MRS, and SMN as well as ModelArts. Set **Policy Name** to **Policy2_AllowOperation** and **Policy View** to **JSON**, configure the policy content, and click **OK**.

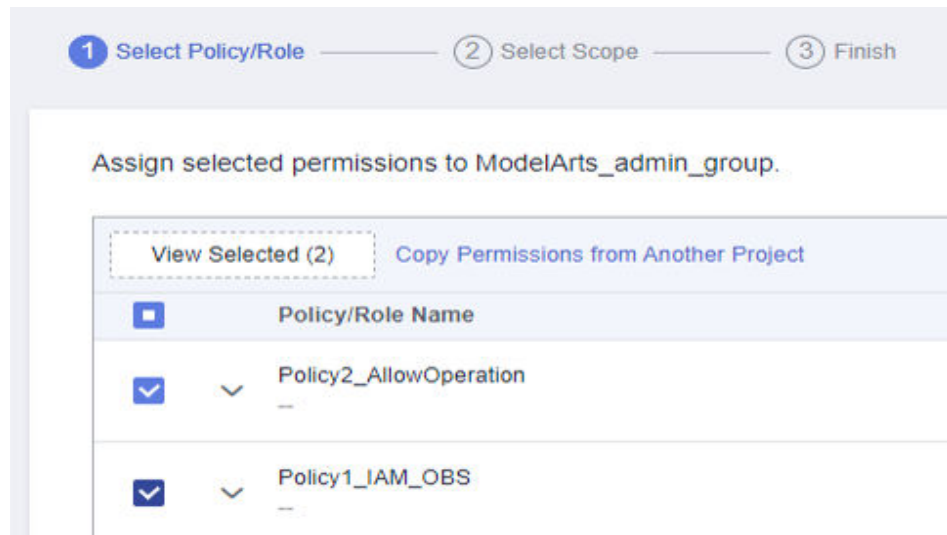
The custom policy **Policy2_AllowOperation** is as follows, which grants the user the permissions to perform operations on dependent services ECS, SWR, MRS, and SMN as well as ModelArts. You can directly copy and paste the content.

```
{
  "Version": "1.1",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecs:serverKeypairs:list",
        "ecs:serverKeypairs:get",
        "ecs:serverKeypairs:delete",
        "ecs:serverKeypairs:create",
        "swr:repository:getNamespace",
        "swr:repository:listNamespaces",
        "swr:repository:deleteTag",
        "swr:repository:getRepository",
        "swr:repository:listTags",
        "swr:instance:createTempCredential",
        "mrs:cluster:get",
        "modelarts:*:*"
      ]
    }
  ]
}
```

Step 3 Grant the policy created in **Step 2** to the administrator group **ModelArts_admin_group**.

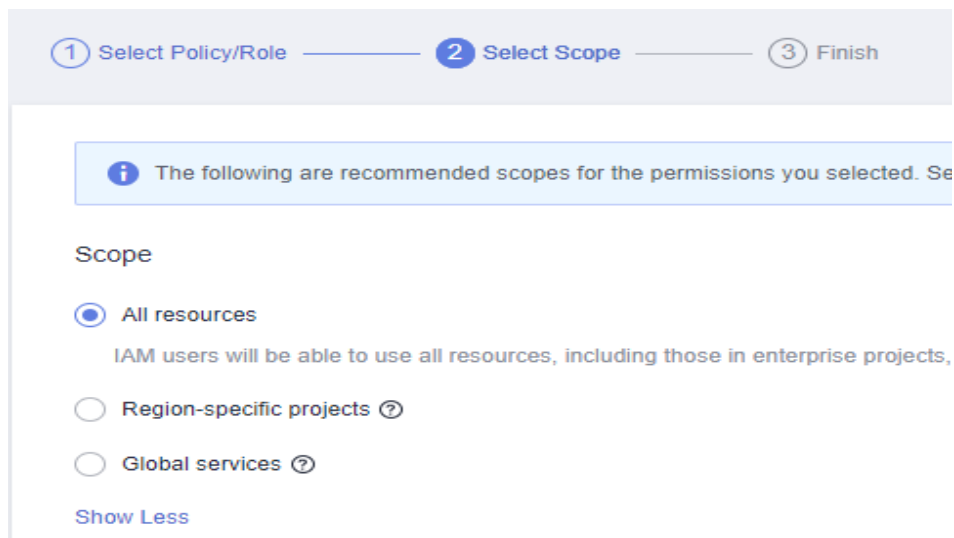
1. In the navigation pane of the IAM console, choose **User Groups**. On the **User Groups** page, locate the row that contains **ModelArts_admin_group**, click **Authorize** in the **Operation** column, and select **Policy1_IAM_OBS** and **Policy2_AllowOperation**. Click **Next**.

Figure 1-13 Select Policy/Role



2. Specify the scope as **All resources** and click **OK**.

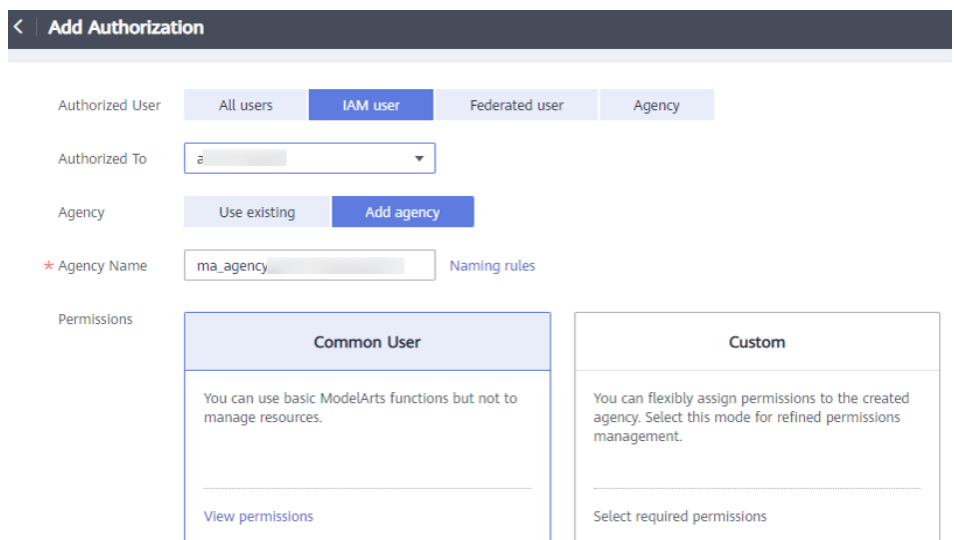
Figure 1-14 Select Scope



Step 4 Configure agent-based ModelArts access authorization for an administrator to allow ModelArts to access dependent services such as OBS.

1. Log in to the ModelArts management console using a tenant account. In the navigation pane, choose **Settings**. The **Global Configuration** page is displayed.
2. Click **Add Authorization**. On the **Add Authorization** page, set **Authorized User** to **IAM user**, select an administrator account for **Authorized To**, select **Add agency**, and select **Common User** for **Permissions**. Permissions control is not required for administrators, so use default setting **Common User**.

Figure 1-15 Configuring authorization for an administrator



3. Click **Create**.

Step 5 Test administrator permissions.

1. Log in to the ModelArts management console as the administrator. On the login page, ensure that **IAM User Login** is selected.
Change the password as prompted upon the first login.

2. In the navigation pane of the ModelArts management console, choose **Dedicated Resource Pools** and click **Create**. If the console does not display a message indicating insufficient permissions, the permissions have been assigned to the administrator.

----End

Configuring Permissions for a Developer

Use IAM for fine-grained control of developer permissions. The procedure is as follows:

Step 1 Use a tenant account to create a developer user group **user_group** and add developer accounts to **user_group**.

Step 2 Create a custom policy.

1. Log in to the management console using a tenant account, hover over your username in the upper right corner, and click **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. Create custom policy 3 to prevent users from performing operations on ModelArts dedicated resource pools and viewing notebook instances of other users.

In the navigation pane of the IAM console, choose **Permissions > Policies/Roles**. Click **Create Custom Policy** in the upper right corner. On the displayed page, enter **Policy3_DenyOperation** for **Policy Name**, select **JSON** for **Policy View**, configure the policy content, and click **OK**.

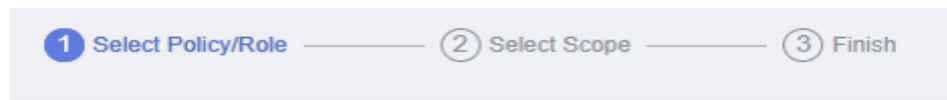
The custom policy **Policy3_DenyOperation** is as follows. You can copy and paste the content.

```
{
  "Version": "1.1",
  "Statement": [
    {
      "Effect": "deny",
      "Action": [
        "modelarts:pool:create",
        "modelarts:pool:update",
        "modelarts:pool:delete",
        "modelarts:notebook:listAllNotebooks"
      ]
    }
  ]
}
```

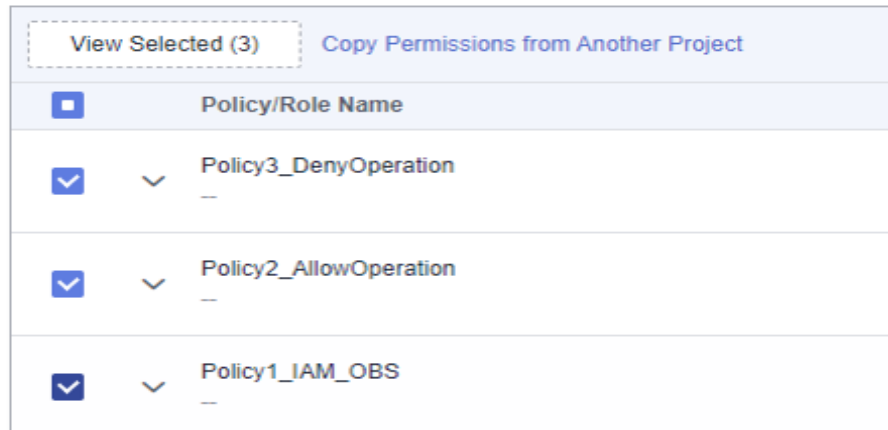
Step 3 Grant the custom policy to the developer user group **user_group**.

1. In the navigation pane of the IAM console, choose **User Groups**. On the **User Groups** page, locate the row that contains **user_group**, click **Authorize** in the **Operation** column, and select **Policy1_IAM_OBS**, **Policy2_AllowOperation**, and **Policy3_DenyOperation**. Click **Next**.

Figure 1-16 Select Policy/Role

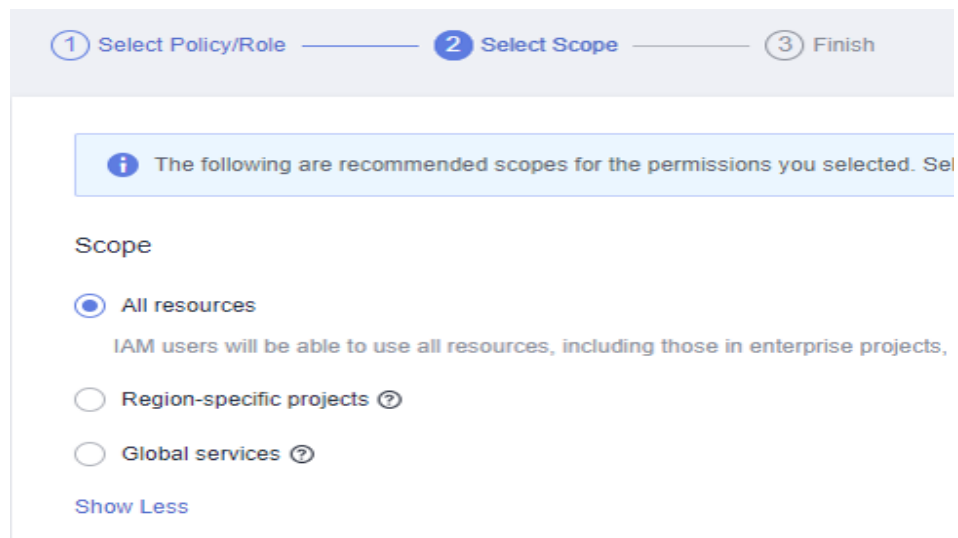


Assign selected permissions to user_group.



2. Specify the scope as **All resources** and click **OK**.

Figure 1-17 Select Scope



Step 4 Configure agent-based ModelArts access authorization for a developer to allow ModelArts to access dependent services such as OBS.

1. Log in to the ModelArts management console using a tenant account. In the navigation pane, choose **Settings**. The **Global Configuration** page is displayed.
2. Click **Add Authorization**. On the **Add Authorization** page, set **Authorized User** to **IAM user**, select a developer account for **Authorized To**, add an agency **ma_agency_develop_user**, set **Permissions** to **Custom**, and select **OBS Administrator**. Developers only need OBS authorization to allow developers to access OBS when using notebook.

Figure 1-18 Configuring authorization for a developer

Authorized User: All users | **IAM user** | Federated user | Agency

Authorized To:

Agency: Use existing | **Add agency**

* Agency Name: Naming rules

Permissions:

Common User

You can use basic ModelArts functions but not to manage resources.

[View permissions](#)

Custom

You can flexibly assign permissions to the created agency. Select this mode for refined permissions management.

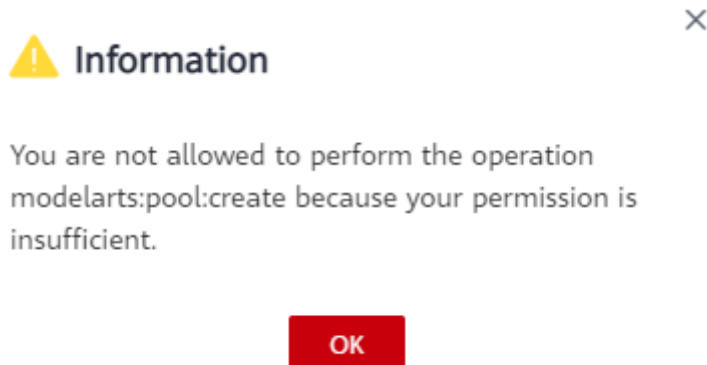
[Select required permissions](#)

Policy/Role	Module	Description
<input checked="" type="checkbox"/> OBS Administrator	Data Management DevEnviron Training Managem...	Object Storage Service Administrator

3. Click **Create**.
4. On the **Global Configuration** page, click **Add Authorization** again. On the **Add Authorization** page that is displayed, configure an agency for other developer users.
On the **Add Authorization** page, set **Authorized User** to **IAM user**, select a developer account for **Authorized To**, and select the existing agency **ma_agency_develop_user** created before.

Step 5 Test developer permissions.

1. Log in to the ModelArts management console as an IAM user in **user_group**. On the login page, ensure that **IAM User Login** is selected.
Change the password as prompted upon the first login.
2. In the navigation pane of the ModelArts management console, choose **Dedicated Resource Pools** and click **Create**. If the console does not display a message indicating insufficient permissions, the permissions have been assigned to the developer.

Figure 1-19 Insufficient permissions

----End

1.3.3 Viewing the Notebook Instances of All IAM Users Under One Tenant Account

Any IAM user granted with the **listAllNotebooks** and **listUsers** permissions can click **View all** on the notebook page to view the instances of all users in the current IAM project.

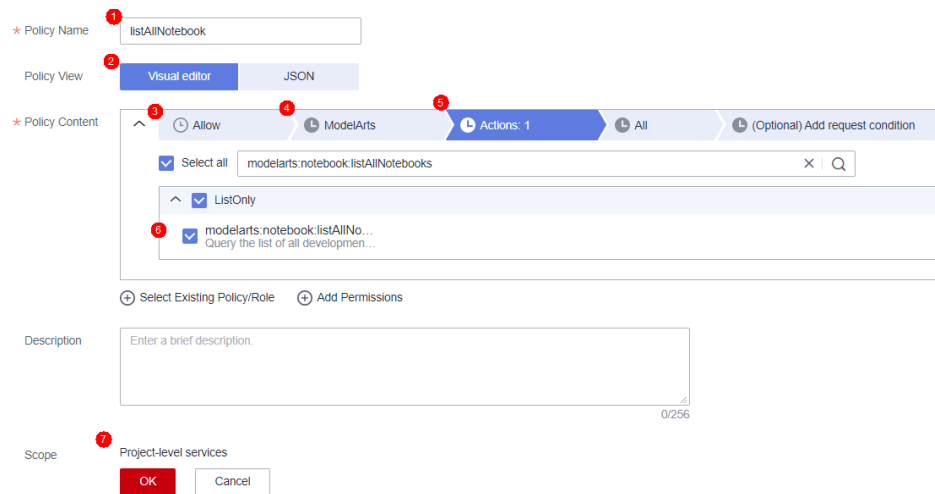
NOTE

Users granted with these permissions can also access OBS and SWR of all users in the current IAM project.

Assigning the Required Permissions

1. Log in to the management console as a tenant user, hover the cursor over your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. On the IAM console, choose **Permissions > Policies/Roles** from the navigation pane, click **Create Custom Policy** in the upper right corner, and create two policies.
Policy 1: Create a policy that allows users to view all notebook instances of an IAM project, as shown in [Figure 1-20](#).
 - **Policy Name:** Enter a custom policy name, for example, **Viewing all notebook instances**.
 - **Policy View:** Select **Visual editor**.
 - **Policy Content:** Select **Allow, ModelArts Service, modelarts:notebook:listAllNotebooks**, and default resources.

Figure 1-20 Creating a custom policy



Policy 2: Create a policy that allows users to view all users of an IAM project.

- **Policy Name:** Enter a custom policy name, for example, **Viewing all users of the current IAM project**.
 - **Policy View:** Select **Visual editor**.
 - **Policy Content:** Select **Allow, Identity and Access Management, iam:users:listUsers**, and default resources.
3. In the navigation pane, choose **User Groups**. On the **User Groups** page, locate the row containing the target user group and click **Authorize** in the **Operation** column. On the **Authorize User Group** page, select the custom policy created in 2 and click **Next**. Then, select the scope and click **OK**.

After the configuration, all users in the user group have the permission to view all notebook instances created by users in the user group.

If no user group is available, create one, add users to it through user group management, and configure authorization for the user group. If the target user is not in a user group, add the user to a user group through user group management.

Enabling an IAM User to Start Other User's Notebook Instance

If an IAM user wants to access another IAM user's notebook instance through remote SSH, they need to update the SSH key pair to their own. Otherwise, error **ModelArts.6786** will be reported. For details about how to update a key pair, see [Modifying the SSH Configuration for a Notebook Instance](#).

ModelArts.6789: Failed to find SSH key pair KeyPair-xxx on the ECS key pair page. Update the key pair and try again later.

1.3.4 Logging In to a Training Container Using Cloud Shell

Application Scenarios

You can use Cloud Shell provided by the ModelArts console to log in to a running training container.

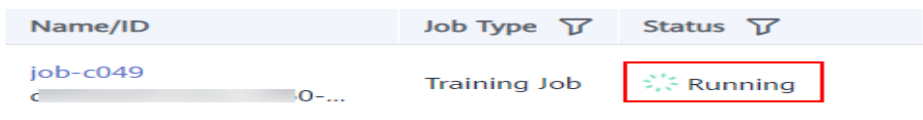
Constraints

You can use Cloud Shell to log in to a running training container using a dedicated resource pool.

Figure 1-21 Selecting a dedicated resource pool when creating a training job



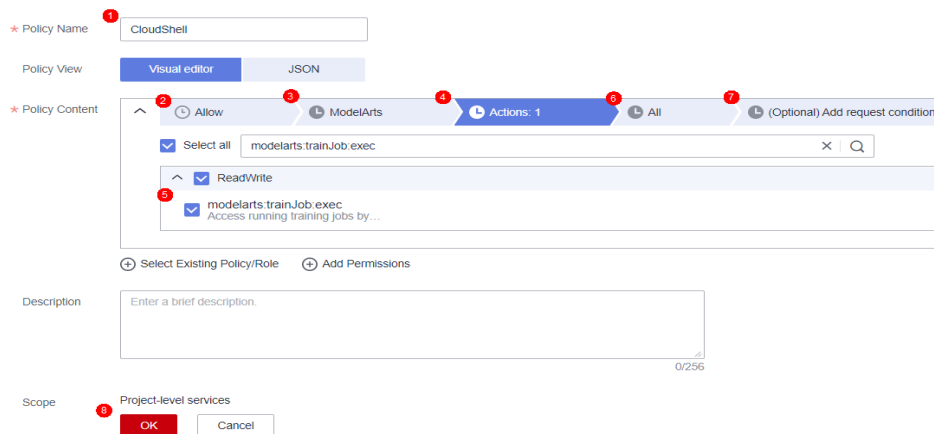
Figure 1-22 A running training job



Preparation: Assigning the Cloud Shell Permission to an IAM User

1. Log in to the management console as a tenant user, hover the cursor over your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. On the IAM console, choose **Permissions > Policies/Roles** from the navigation pane, click **Create Custom Policy** in the upper right corner, and configure the following parameters.
 - **Policy Name:** Enter a custom policy name, for example, **Using Cloud Shell to log in to a running training container**.
 - **Policy View:** Select **Visual editor**.
 - **Policy Content:** Select **Allow**, **ModelArts Service**, **modelarts:trainJob:exec**, and default resources.

Figure 1-23 Creating a custom policy



3. In the navigation pane, choose **User Groups**. Then, click **Authorize** in the **Operation** column of the target user group. On the **Authorize User Group** page, select the custom policies created in 2, and click **Next**. Then, select the scope and click **OK**.

After the configuration, all users in the user group have the permission to use Cloud Shell to log in to a running training container.

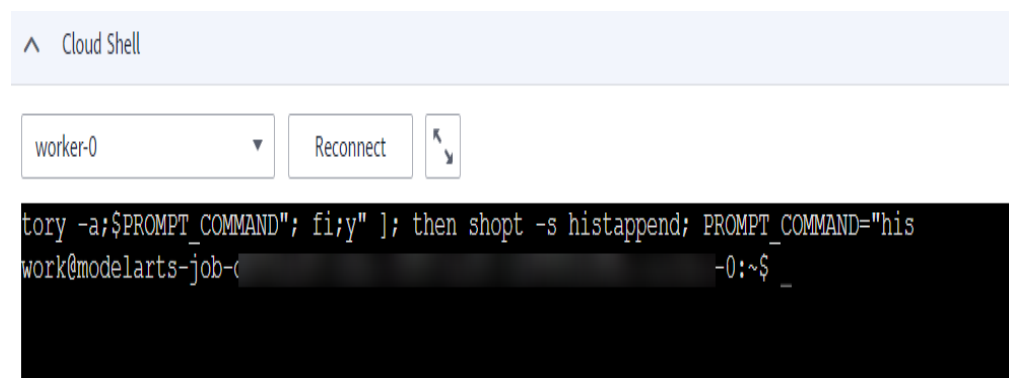
If no user group is available, create one, add users to it through user group management, and configure authorization for the user group. If the target user is not in a user group, add the user to a user group through user group management.

Using Cloud Shell

1. Configure parameters based on [Preparation: Assigning the Cloud Shell Permission to an IAM User](#).
2. On the ModelArts console, choose **Training Management > Training Jobs**. Go to the details page of the target training job and log in to the training container on the Cloud Shell tab.

Verify that the login is successful, as shown in the following figure.

Figure 1-24 Cloud Shell



1.3.5 Prohibiting a User from Using a Public Resource Pool

This section describes how to control the ModelArts permissions of a user so that the user is not allowed to use a public resource pool to create training jobs, create notebook instances, or deploy inference services.

Context

Through permission control, ModelArts dedicated resource pool users can be prohibited from using a public resource pool to create training jobs, create notebook instances, or deploy inference services.

To control the permissions, configure the following permission policy items:

- **modelarts:notebook:create**: allows you to create a notebook instance.
- **modelarts:trainJob:create**: allows you to create a training job.
- **modelarts:service:create**: allows you to create an inference service.

Procedure

1. Log in to the management console as a tenant user, hover the cursor over your username in the upper right corner, and choose **Identity and Access Management** from the drop-down list to switch to the IAM management console.
2. In the navigation pane, choose **Permissions > Policies/Roles**. On the **Policies/Roles** page, click **Create Custom Policy** in the upper right corner, configure parameters, and click **OK**.
 - **Policy Name**: Configure the policy name.
 - **Policy View**: Select **Visual editor** or **JSON**.
 - **Policy Content**: Select **Deny**. In **Select service**, search for **ModelArts** and select it. In **ReadWrite** under **Actions**, search for **modelarts:trainJob:create**, **modelarts:notebook:create**, and **modelarts:service:create** and select them. **All**: Retain the default setting. In **Add request condition**, click **Add Request Condition**. In the displayed dialog box, set **Condition Key** to **modelarts:poolType**, **Operator** to **StringEquals**, and **Value** to **public**.

The policy content in JSON view is as follows:

```
{
  "Version": "1.1",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "modelarts:trainJob:create",
        "modelarts:notebook:create",
        "modelarts:service:create"
      ],
      "Condition": {
        "StringEquals": {
          "modelarts:poolType": [
            "public"
          ]
        }
      }
    }
  ]
}
```

3. In the navigation pane, choose **User Groups**. On the **User Groups** page, locate the row containing the target user group and click **Authorize** in the **Operation** column. On the **Authorize User Group** page, select the custom policy created in 2 and click **Next**. Then, select the scope and click **OK**.

After the configuration, all users in the user group have the permission to view all notebook instances created by users in the user group.

If no user group is available, create one, add users to it through user group management, and configure authorization for the user group. If the target user is not in a user group, add the user to a user group through user group management.
4. Add the policy to the user's agency authorization. This prevents the user from breaking the permission scope through a token on the tenant plane.

In the navigation pane, choose **Agencies**. Locate the agency used by the user group on ModelArts and click **Modify** in the **Operation** column. On the **Permissions** tab page, click **Authorize**, select the created custom policy, and click **Next**. Select the scope for authorization and click **OK**.

Verification

Log in to the ModelArts console as an IAM user, choose **Training Management > Training Jobs**, and click **Create Training Job**. On the page for creating a training job, only a dedicated resource pool can be selected for **Resource Pool**.

Log in to the ModelArts console as an IAM user, choose **DevEnviron > Notebook**, and click **Create**. On the page for creating a notebook instance, only a dedicated resource pool can be selected for **Resource Pool**.

Log in to the ModelArts console as an IAM user, choose **Service Deployment > Real-Time Services**, and click **Deploy**. On the page for service deployment, only a dedicated resource pool can be selected for **Resource Pool**.

2 Model Development (Custom Algorithms in Training Jobs of the New Version)

2.1 Using a Custom Algorithm to Build a Handwritten Digit Recognition Model

This section describes how to modify a local custom algorithm to train and deploy models on ModelArts.

Scenarios

This case describes how to use PyTorch 1.8 to recognize handwritten digit images. An official MNIST dataset is used in this case.

Through this case, you can learn how to train jobs, deploy an inference model, and perform prediction on ModelArts.

Process

Before performing the following operations, complete necessary operations. For details, see [Preparations](#).

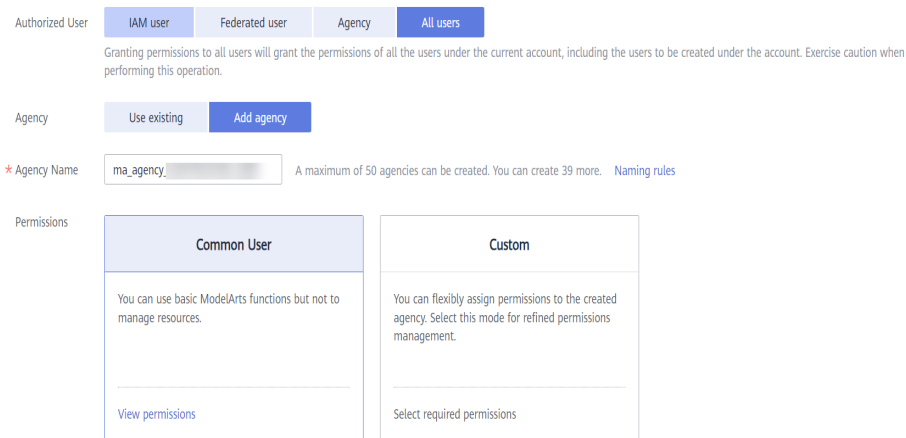
1. **Step 1 Prepare the Training Data:** Download the MNIST dataset.
2. **Step 2 Prepare Training Files and Inference Files:** Write training and inference code.
3. **Step 3 Create an OBS Bucket and Upload Files to OBS:** Create an OBS bucket and folder, and upload the dataset, training script, inference script, and inference configuration file to OBS.
4. **Step 4 Create a Training Job:** Train a model.
5. **Step 5 Deploy the Model for Inference:** Import the trained model to ModelArts, create an AI application, and deploy the AI application as a real-time service.

6. **Step 6 Perform Prediction:** Upload a handwritten digit image and send an inference request to obtain the inference result.
7. **Step 7 Release Resources:** Stop the service and delete the data in OBS to stop billing.

Preparations

- You have registered a Huawei ID and enabled Huawei Cloud services, and the account is not in arrears or frozen.
- You have configured the agency-based authorization.
Certain ModelArts functions require access to OBS, SWR, and IEF. Before using ModelArts, ensure your account has been authorized to access these services.
 - a. Log in to the ModelArts console using your Huawei Cloud account. In the navigation pane on the left, choose **Settings**. Then, on the **Global Configuration** page, click **Add Authorization**.
 - b. On the **Add Authorization** page that is displayed, set required parameters as follows:
Authorized User: Select **All users**.
Agency: Select **Add agency**.
Permissions: Select **Common User**.
 Select "I have read and agree to the ModelArts Service Statement", and click **Create**.

Figure 2-1 Configuring the agency-based authorization



- c. After the configuration is complete, view the agency configurations of your account on the **Global Configuration** page.

Figure 2-2 Viewing agency configurations

Authorized To	Authorized User	Authorization Type	Authorization Content	Creation Time	Operation
All users	All users	Agency	ma_agency_	Sep 15, 2023 17:29:30 GMT+08:00	View Permissions Delete

Step 1 Prepare the Training Data

An MNIST dataset downloaded from the [MNIST official website](#) is used in this case. Ensure that the four files are all downloaded.

Figure 2-3 MNIST dataset

Four files are available on this site:

[train-images-idx3-ubyte.gz](#): training set images (9912422 bytes)
[train-labels-idx1-ubyte.gz](#): training set labels (28881 bytes)
[t10k-images-idx3-ubyte.gz](#): test set images (1648877 bytes)
[t10k-labels-idx1-ubyte.gz](#): test set labels (4542 bytes)

- **train-images-idx3-ubyte.gz**: compressed package of the training set, which contains 60,000 samples.
- **train-labels-idx1-ubyte.gz**: compressed package of the training set labels, which contains the labels of the 60,000 samples
- **t10k-images-idx3-ubyte.gz**: compressed package of the validation set, which contains 10,000 samples.
- **t10k-labels-idx1-ubyte.gz**: compressed package of the validation set labels, which contains the labels of the 10,000 samples

NOTE

If you are asked to enter the login information after you click the MNIST official website link, copy and paste this link in the address box of your browser: <http://yann.lecun.com/exdb/mnist/>

The login information is required when you open the link in HTTPS mode, which is not required if you open the link in HTTP mode.

Step 2 Prepare Training Files and Inference Files

In this case, ModelArts provides the training script, inference script, and inference configuration file.

NOTE

When pasting code from a .py file, create a .py file. Otherwise, the error message "SyntaxError: 'gbk' codec can't decode byte 0xa4 in position 324: illegal multibyte sequence" may be displayed.

Create the training script **train.py** on the local host. The content is as follows:

```
# base on https://github.com/pytorch/examples/blob/main/mnist/main.py

from __future__ import print_function

import os
import gzip
import codecs
import argparse
from typing import IO, Union

import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR

import shutil
```

```
# Define a network model.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

# Train the model. Set the model to the training mode, load the training data, calculate the loss function,
and perform gradient descent.
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{} / {}] {:.0f}%] \tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if args.dry_run:
                break

# Validate the model. Set the model to the validation mode, load the validation data, and calculate the loss
function and accuracy.
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {} / {} {:.0f}%\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```



```
# The following is PyTorch MNIST.
# https://github.com/pytorch/vision/blob/v0.9.0/torchvision/datasets/mnist.py
def get_int(b: bytes) -> int:
    return int(codecs.encode(b, 'hex'), 16)

def open_maybe_compressed_file(path: Union[str, IO]) -> Union[IO, gzip.GzipFile]:
    """Return a file object that possibly decompresses 'path' on the fly.
    Decompression occurs when argument 'path' is a string and ends with '.gz' or '.xz'.
    """
    if not isinstance(path, torch._six.string_classes):
        return path
    if path.endswith('.gz'):
        return gzip.open(path, 'rb')
    if path.endswith('.xz'):
        return lzma.open(path, 'rb')
    return open(path, 'rb')

SN3_PASCALVINCENT_TYEMAP = {
    8: (torch.uint8, np.uint8, np.uint8),
    9: (torch.int8, np.int8, np.int8),
    11: (torch.int16, np.dtype('>i2'), 'i2'),
    12: (torch.int32, np.dtype('>i4'), 'i4'),
    13: (torch.float32, np.dtype('>f4'), 'f4'),
    14: (torch.float64, np.dtype('>f8'), 'f8')
}

def read_sn3_pascalvincent_tensor(path: Union[str, IO], strict: bool = True) -> torch.Tensor:
    """Read a SN3 file in "Pascal Vincent" format (Lush file 'libidx/idx-io.lsh').
    Argument may be a filename, compressed filename, or file object.
    """
    # read
    with open_maybe_compressed_file(path) as f:
        data = f.read()
    # parse
    magic = get_int(data[0:4])
    nd = magic % 256
    ty = magic // 256
    assert 1 <= nd <= 3
    assert 8 <= ty <= 14
    m = SN3_PASCALVINCENT_TYEMAP[ty]
    s = [get_int(data[4 * (i + 1): 4 * (i + 2)]) for i in range(nd)]
    parsed = np.frombuffer(data, dtype=m[1], offset=(4 * (nd + 1)))
    assert parsed.shape[0] == np.prod(s) or not strict
    return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)

def read_label_file(path: str) -> torch.Tensor:
    with open(path, 'rb') as f:
        x = read_sn3_pascalvincent_tensor(f, strict=False)
        assert(x.dtype == torch.uint8)
        assert(x.ndimension() == 1)
        return x.long()

def read_image_file(path: str) -> torch.Tensor:
    with open(path, 'rb') as f:
        x = read_sn3_pascalvincent_tensor(f, strict=False)
        assert(x.dtype == torch.uint8)
        assert(x.ndimension() == 3)
        return x

def extract_archive(from_path, to_path):
    to_path = os.path.join(to_path, os.path.splitext(os.path.basename(from_path))[0])
    with open(to_path, "wb") as out_f, gzip.GzipFile(from_path) as zip_f:
        out_f.write(zip_f.read())
```

```

# The above is pytorch mnist.
# --- end

# Raw MNIST dataset processing
def convert_raw_mnist_dataset_to_pytorch_mnist_dataset(data_url):
    """
    raw

    {data_url}/
    train-images-idx3-ubyte.gz
    train-labels-idx1-ubyte.gz
    t10k-images-idx3-ubyte.gz
    t10k-labels-idx1-ubyte.gz

    processed

    {data_url}/
    train-images-idx3-ubyte.gz
    train-labels-idx1-ubyte.gz
    t10k-images-idx3-ubyte.gz
    t10k-labels-idx1-ubyte.gz
    MNIST/raw
    train-images-idx3-ubyte
    train-labels-idx1-ubyte
    t10k-images-idx3-ubyte
    t10k-labels-idx1-ubyte
    MNIST/processed
    training.pt
    test.pt
    """
    resources = [
        "train-images-idx3-ubyte.gz",
        "train-labels-idx1-ubyte.gz",
        "t10k-images-idx3-ubyte.gz",
        "t10k-labels-idx1-ubyte.gz"
    ]

    pytorch_mnist_dataset = os.path.join(data_url, 'MNIST')

    raw_folder = os.path.join(pytorch_mnist_dataset, 'raw')
    processed_folder = os.path.join(pytorch_mnist_dataset, 'processed')

    os.makedirs(raw_folder, exist_ok=True)
    os.makedirs(processed_folder, exist_ok=True)

    print('Processing...')

    for f in resources:
        extract_archive(os.path.join(data_url, f), raw_folder)

    training_set = (
        read_image_file(os.path.join(raw_folder, 'train-images-idx3-ubyte')),
        read_label_file(os.path.join(raw_folder, 'train-labels-idx1-ubyte'))
    )
    test_set = (
        read_image_file(os.path.join(raw_folder, 't10k-images-idx3-ubyte')),
        read_label_file(os.path.join(raw_folder, 't10k-labels-idx1-ubyte'))
    )
    with open(os.path.join(processed_folder, 'training.pt'), 'wb') as f:
        torch.save(training_set, f)
    with open(os.path.join(processed_folder, 'test.pt'), 'wb') as f:
        torch.save(test_set, f)

    print('Done!')

def main():
    # Define the preset running parameters of the training job.

```

```
parser = argparse.ArgumentParser(description='PyTorch MNIST Example')

parser.add_argument('--data_url', type=str, default=False,
                    help='mnist dataset path')
parser.add_argument('--train_url', type=str, default=False,
                    help='mnist model path')

parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                    help='input batch size for training (default: 64)')
parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                    help='input batch size for testing (default: 1000)')
parser.add_argument('--epochs', type=int, default=14, metavar='N',
                    help='number of epochs to train (default: 14)')
parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                    help='learning rate (default: 1.0)')
parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                    help='Learning rate step gamma (default: 0.7)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
parser.add_argument('--dry-run', action='store_true', default=False,
                    help='quickly check a single pass')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')
parser.add_argument('--save-model', action='store_true', default=True,
                    help='For Saving the current Model')
args = parser.parse_args()

use_cuda = not args.no_cuda and torch.cuda.is_available()

torch.manual_seed(args.seed)

# Set whether to use GPU or CPU to run the algorithm.
device = torch.device("cuda" if use_cuda else "cpu")

train_kwargs = {'batch_size': args.batch_size}
test_kwargs = {'batch_size': args.test_batch_size}
if use_cuda:
    cuda_kwargs = {'num_workers': 1,
                  'pin_memory': True,
                  'shuffle': True}
    train_kwargs.update(cuda_kwargs)
    test_kwargs.update(cuda_kwargs)

# Define the data preprocessing method.
transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Convert the raw MNIST dataset to a PyTorch MNIST dataset.
convert_raw_mnist_dataset_to_pytorch_mnist_dataset(args.data_url)

# Create a training dataset and a validation dataset.
dataset1 = datasets.MNIST(args.data_url, train=True, download=False,
                          transform=transform)
dataset2 = datasets.MNIST(args.data_url, train=False, download=False,
                          transform=transform)

# Create iterators for the training dataset and the validation dataset.
train_loader = torch.utils.data.DataLoader(dataset1, **train_kwargs)
test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)

# Initialize the neural network model and copy the model to the compute device.
model = Net().to(device)
# Define the training optimizer and learning rate for gradient descent calculation.
optimizer = optim.Adadelta(model.parameters(), lr=args.lr)
scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
```

```
# Train the neural network and perform validation in each epoch.
for epoch in range(1, args.epochs + 1):
    train(args, model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
    scheduler.step()

# Save the model and make it adapted to the ModelArts inference model package specifications.
if args.save_model:

    # Create the model directory in the path specified in train_url.
    model_path = os.path.join(args.train_url, 'model')
    os.makedirs(model_path, exist_ok = True)

    # Save the model to the model directory based on the ModelArts inference model package
    specifications.
    torch.save(model.state_dict(), os.path.join(model_path, 'mnist_cnn.pt'))

    # Copy the inference code and configuration file to the model directory.
    the_path_of_current_file = os.path.dirname(__file__)
    shutil.copyfile(os.path.join(the_path_of_current_file, 'infer/customize_service.py'),
os.path.join(model_path, 'customize_service.py'))
    shutil.copyfile(os.path.join(the_path_of_current_file, 'infer/config.json'), os.path.join(model_path,
'config.json'))

if __name__ == '__main__':
    main()
```

Create the inference script **customize_service.py** on the local host. The content is as follows:

```
import os
import log
import json

import torch.nn.functional as F
import torch.nn as nn
import torch
import torchvision.transforms as transforms

import numpy as np
from PIL import Image

from model_service.pytorch_model_service import PTServingBaseService

logger = log.getLogger(__name__)

# Define model preprocessing.
infer_transformation = transforms.Compose([
    transforms.Resize(28),
    transforms.CenterCrop(28),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Model inference service
class PTVisionService(PTServingBaseService):

    def __init__(self, model_name, model_path):
        # Call the constructor of the parent class.
        super(PTVisionService, self).__init__(model_name, model_path)

        # Call the customized function to load the model.
        self.model = Mnist(model_path)

        # Load labels.
        self.label = [0,1,2,3,4,5,6,7,8,9]

    # Receive the request data and convert it to the input format acceptable to the model.
    def _preprocess(self, data):
```

```

preprocessed_data = {}
for k, v in data.items():
    input_batch = []
    for file_name, file_content in v.items():
        with Image.open(file_content) as image1:
            # Gray processing
            image1 = image1.convert("L")
            if torch.cuda.is_available():
                input_batch.append(infer_transformation(image1).cuda())
            else:
                input_batch.append(infer_transformation(image1))
    input_batch_var = torch.autograd.Variable(torch.stack(input_batch, dim=0), volatile=True)
    print(input_batch_var.shape)
    preprocessed_data[k] = input_batch_var

return preprocessed_data

# Post-process the inference result to obtain the expected output format. The result is the returned value.
def _postprocess(self, data):
    results = []
    for k, v in data.items():
        result = torch.argmax(v[0])
        result = {k: self.label[result]}
        results.append(result)
    return results

# Perform forward inference on the input data to obtain the inference result.
def _inference(self, data):
    result = {}
    for k, v in data.items():
        result[k] = self.model(v)

    return result

# Define a network.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

def Mnist(model_path, **kwargs):
    # Generate a network.
    model = Net()

    # Load the model.
    if torch.cuda.is_available():
        device = torch.device('cuda')

```

```
model.load_state_dict(torch.load(model_path, map_location="cuda:0"))
else:
    device = torch.device('cpu')
    model.load_state_dict(torch.load(model_path, map_location=device))

# CPU or GPU mapping
model.to(device)

# Turn the model to inference mode.
model.eval()

return model
```

Infer the configuration file **config.json** on the local host. The content is as follows:

```
{
  "model_algorithm": "image_classification",
  "model_type": "PyTorch",
  "runtime": "pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64"
}
```

Step 3 Create an OBS Bucket and Upload Files to OBS

Upload the data, code file, inference code file, and inference configuration file obtained in the previous step to an OBS bucket. When running a training job on ModelArts, read data and code files from the OBS bucket.

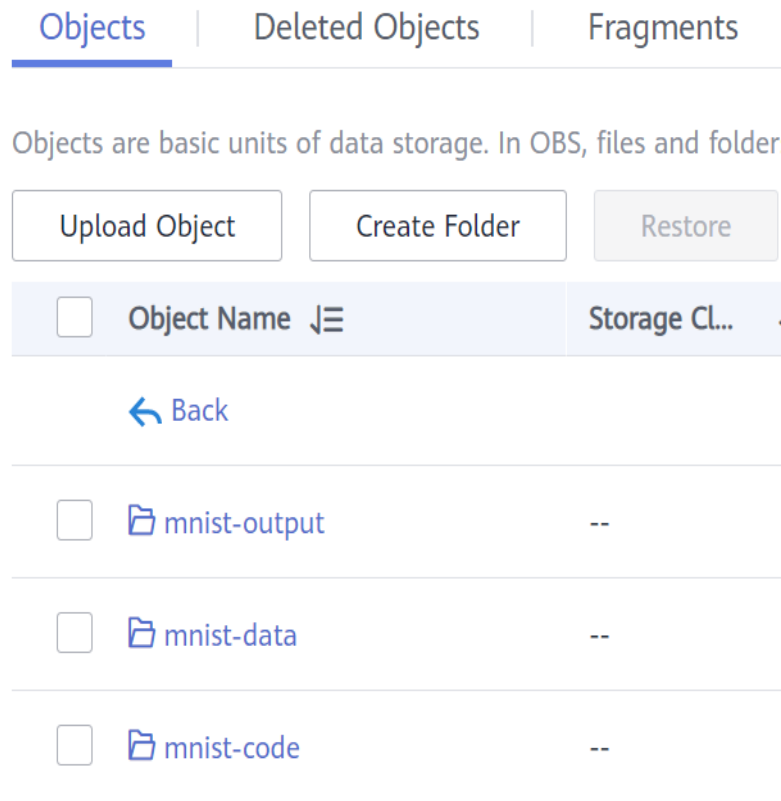
1. Log in to the OBS console and create an OBS bucket and folder. [Figure 2-4](#) shows an example of the created objects. For details, see [and](#) .

```
{OBS bucket}          # OBS bucket name, which is customizable, for example, test-modelarts-xx
  -{OBS folder}      # OBS folder name, which is customizable, for example, pytorch
    - mnist-data    # OBS folder, which is used to store the training dataset. The folder name is
                    # customizable, for example, mnist-data.
    - mnist-code    # OBS folder, which is used to store training script train.py. The folder name is
                    # customizable, for example, mnist-code.
    - infer         # OBS folder, which is used to store inference script customize_service.py and
                    # configuration file config.json
    - mnist-output  # OBS folder, which is used to store trained models. The folder name is
                    # customizable, for example, mnist-output.
```

CAUTION

- The region where the created OBS bucket resides must be the same as that where ModelArts is used. Otherwise, the OBS bucket will be unavailable for training.
 - When creating an OBS bucket, do not set the archive storage class. Otherwise, training models will fail.
-

Figure 2-4 OBS file directory

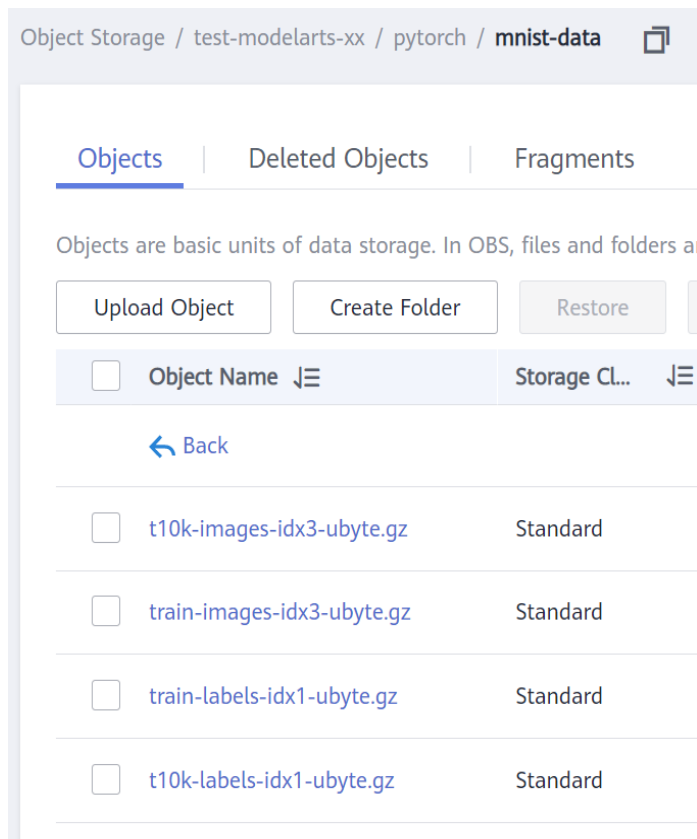


2. Upload the MNIST dataset package obtained in [Step 1 Prepare the Training Data](#) to OBS. For details, see .

CAUTION

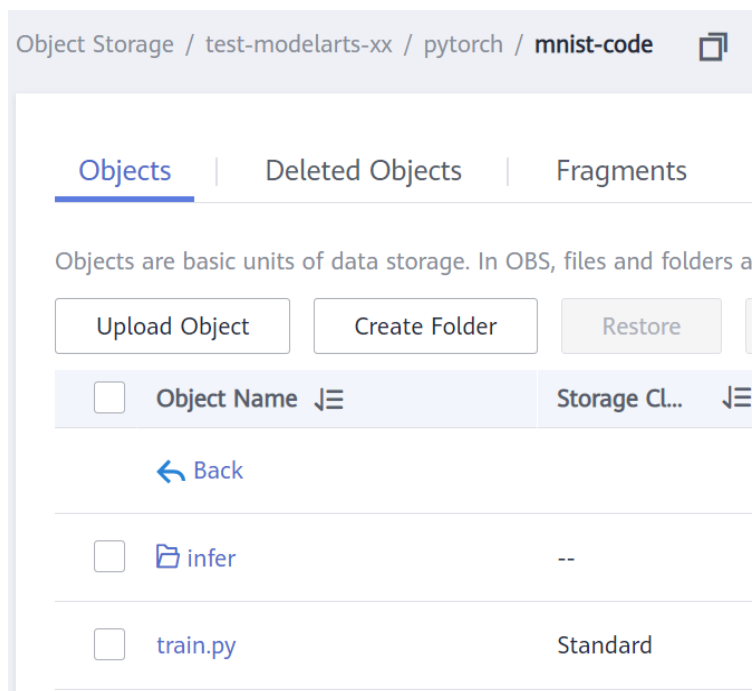
- When uploading data to OBS, do not encrypt the data. Otherwise, the training will fail.
- Files do not need to be decompressed. Directly upload compressed packages to OBS.

Figure 2-5 Uploading a dataset to the mnist-data folder



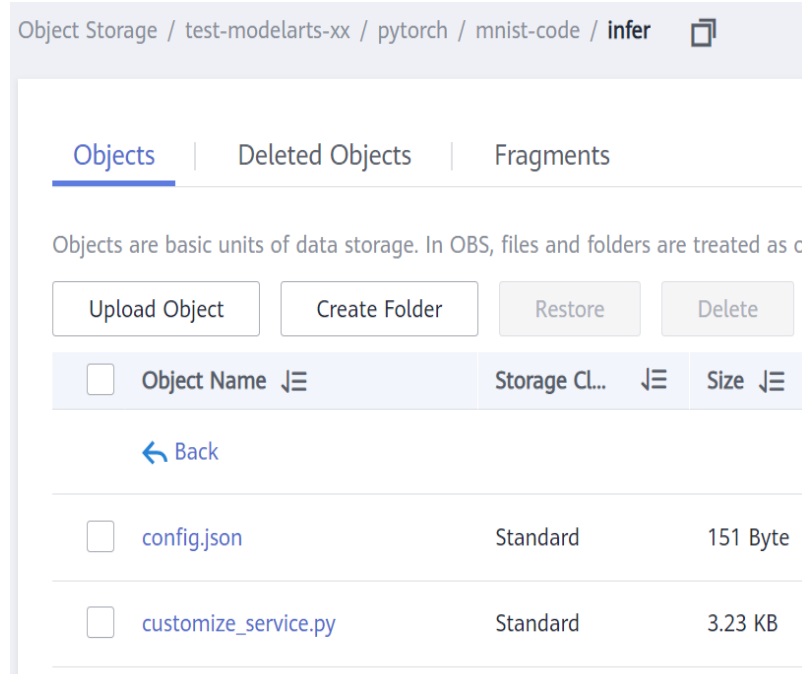
3. Upload the training script **train.py** to the **mnist-code** folder.

Figure 2-6 Uploading the training script train.py to the mnist-code folder



4. Upload the inference script **customize_service.py** and inference configuration file **config.json** to the **infer** folder.

Figure 2-7 Uploading **customize_service.py** and **config.json** to the **infer** folder



Step 4 Create a Training Job

1. Log in to the ModelArts management console and select the same region as the OBS bucket.
2. In the navigation pane on the left, choose **Settings** and check whether access authorization has been configured for the current account. For details, see . If you have been authorized using access keys, clear the authorization and configure agency authorization.
3. In the navigation pane on the left, choose **Training Management > Training Jobs**. On the displayed page, click **Create Training Job**.

Figure 2-8 Training Jobs



4. Set parameters.
 - **Algorithm Type:** Select **Custom algorithm**.
 - **Boot Mode:** Select **Preset image** and then select **PyTorch** and **pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64** from the drop-down lists.
 - **Code Directory:** Select the created OBS code directory, for example, **/test-modelarts-xx/pytorch/mnist-code/** (replace **test-modelarts-xx** with your OBS bucket name).
 - **Boot File:** Select the training script **train.py** uploaded to the code directory.
 - **Input:** Add one input and set its name to **data_url**. Set the data path to your OBS directory, for example, **/test-modelarts-xx/pytorch/mnist-data/** (replace **test-modelarts-xx** with your OBS bucket name).
 - **Output:** Add one output and set its name to **train_url**. Set the data path to your OBS directory, for example, **/test-modelarts-xx/pytorch/mnist-output/** (replace **test-modelarts-xx** with your OBS bucket name). Do not pre-download to a local directory.
 - **Resource Type:** Select **GPU** and then **GPU: 1*NVIDIA-V100(16GB) | CPU: 8 vCPUs 64GB** (example). If there are free GPU specifications, you can select them for training.
 - Retain default settings for other parameters.

NOTE

The sample code runs on a single node with a single card. If you select a flavor with multiple GPUs, the training will fail.

Figure 2-9 Training job settings

Figure 2-10 Setting training input and output

Figure 2-11 Configuring the resource type

5. Click **Submit**, confirm parameter settings for the training job, and click **Yes**. The system automatically switches back to the **Training Jobs** page. When the training job status changes to **Completed**, the model training is completed.

NOTE

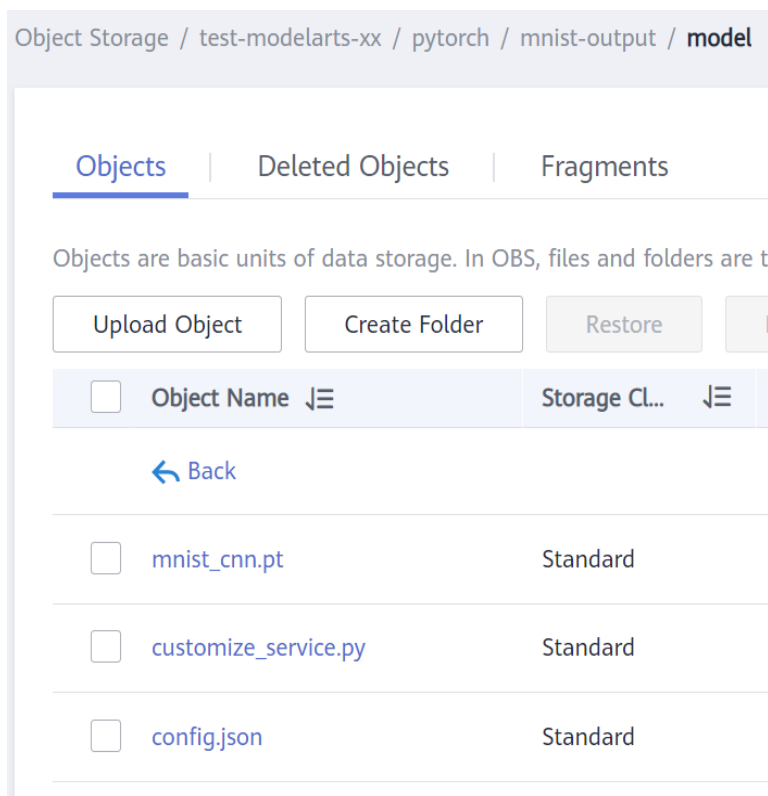
In this case, the training job will take about 10 minutes.

6. Click the training job name. On the job details page that is displayed, check whether there are error messages in logs. If so, the training failed. Identify the cause and locate the fault based on the logs.
7. In the lower left corner of the training details page, click the training output path to go to OBS (as shown in **Figure 2-12**). Then, check whether the **model** folder is available and whether there are any trained models in the folder (as shown in **Figure 2-13**). If there is no **model** folder or trained model, the training input may be incomplete. In this case, completely upload the training data and train the model again.

Figure 2-12 Output path

Training Input			
Input Path	Paramete...	Obtained ...	Local Path (Traini...
/test-modelarts-x...	data_url	Hyperpar...	/home/ma-us...
Training Output			
Output Path	Paramete...	Obtained ...	Local Path (Traini...
/test-modelarts-x...	train_url	Hyperpar...	/home/ma-us...

Figure 2-13 Trained model



Step 5 Deploy the Model for Inference

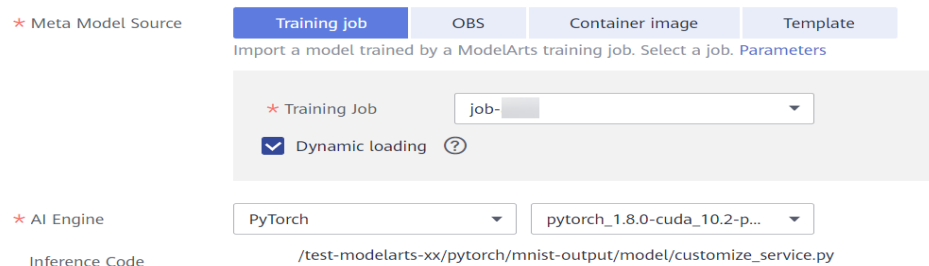
After the model training is complete, create an AI application and deploy it as a real-time service.

1. Log in to the ModelArts management console. In the navigation pane on the left, choose **AI Application Management > AI Applications**. On the **My AI Applications** page, click **Create**.
2. On the **Create** page, configure parameters and click **Create now**.

Choose **Training Job** for **Meta Model Source**. Select the training job completed in **Step 4 Create a Training Job** from the drop-down list and

select **Dynamic loading**. The values of **AI Engine** will be automatically configured.

Figure 2-14 Meta Model Source

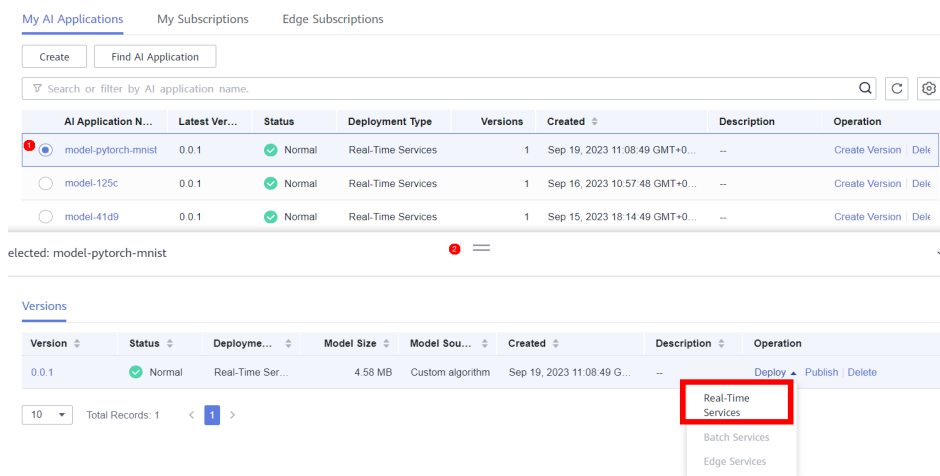


NOTE

If you have used **Training Jobs** of an old version, you can see both **Training Jobs** and **Training Jobs New** below **Training job**. In this case, select **Training Jobs New**.

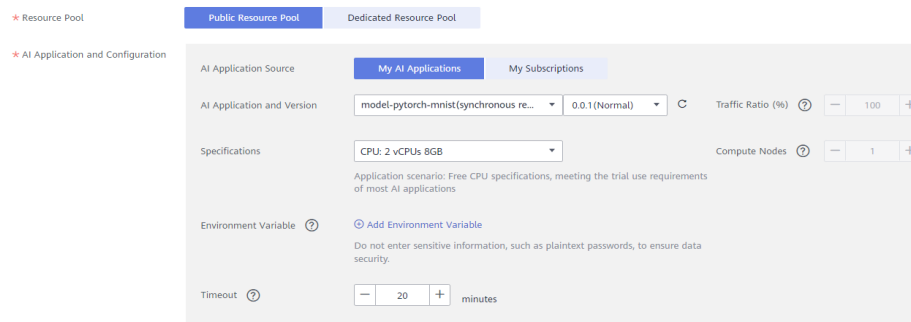
- On the **AI Applications** page, if the application status changes to **Normal**, it has been created. Click the option button on the left of the AI application name to display the version list at the bottom of the list page, and choose **Deploy > Real-Time Services** in the **Operation** column to deploy the AI application as a real-time service.

Figure 2-15 Deploying a real-time service



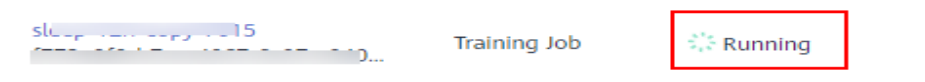
- On the **Deploy** page, configure parameters and create a real-time service as prompted. In this example, use CPU specifications. If there are free CPU specifications, you can select them for deployment. (Each user can deploy only one real-time service for free. If you have deployed one, delete it first before deploying a new one for free.)

Figure 2-16 Deploying a model



After you submit the service deployment request, the system automatically switches to the **Real-Time Services** page. When the service status changes to **Running**, the service has been deployed.

Figure 2-17 Deployed service



Step 6 Perform Prediction

1. On the **Real-Time Services** page, click the name of the real-time service. The real-time service details page is displayed.
2. Click the **Prediction** tab, set **Request Type** to **multipart/form-data**, **Request Parameter** to **image**, click **Upload** to upload a sample image, and click **Predict**.

After the prediction is complete, the prediction result is displayed in the **Test Result** pane. According to the prediction result, the digit on the image is **2**.

NOTE

The MNIST used in this case is a simple dataset used for demonstration, and its algorithms are also simple neural network algorithms used for teaching. The models generated using such data and algorithms are applicable only to teaching but not to complex prediction scenarios. The prediction is accurate only if the image used for prediction is similar to the image in the training dataset (white characters on black background).

Figure 2-18 Example

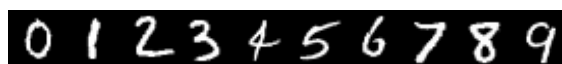
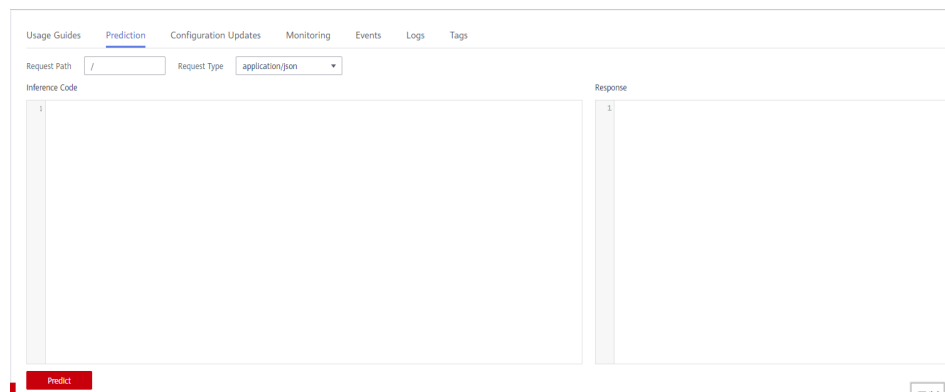


Figure 2-19 Prediction results



Step 7 Release Resources

If you do not need to use this model and real-time service anymore, release the resources to stop billing.

- On the **Real-Time Services** page, locate the row containing the target service and click **Stop** or **Delete** in the **Operation** column.
- On the **AI Applications** page in **AI Application Management**, locate the row containing the target service and click **Delete** in the **Operation** column.
- On the **Training Jobs** page, click **Delete** in the **Operation** column to delete the finished training job.
- Go to OBS and delete the OBS bucket, folders, and files used in this example.

FAQs

- **Why Is a Training Job Always Queuing?**
If the training job is always queuing, the selected resources are limited in the resource pool, and the job needs to be queued. In this case, wait for resources. For details, see .
- **Why Can't I Find My Created OBS Bucket After I Select an OBS Path in ModelArts?**
Ensure that the created bucket is in the same region as ModelArts. For details, see .

3 Model Inference

3.1 Creating a Custom Image and Using It to Create an AI Application

If you want to use an AI engine that is not supported by ModelArts, create a custom image for the engine, import the image to ModelArts, and use the image to create AI applications. This section describes how to use a custom image to create an AI application and deploy the application as a real-time service.

The process is as follows:

1. **Building an Image Locally:** Create a custom image package locally. For details, see [Custom Image Specifications for Creating AI Applications](#).
2. **Verifying the Image Locally and Uploading It to SWR:** Verify the APIs of the custom image and upload the custom image to SWR.
3. **Using the Custom Image to Create an AI Application:** Import the image to ModelArts AI application management.
4. **Deploying the AI Application as a Real-Time Service:** Deploy the model as a real-time service.

Building an Image Locally

This section uses a Linux x86_x64 host as an example. You can use an existing local host to create a custom image.

1. Install Docker. For details, see [Docker official documents](#). The following shows an example:

```
curl -fsSL get.docker.com -o get-docker.sh
sh get-docker.sh
```
2. Obtain the base image. Ubuntu 18.04 is used in this example.

```
docker pull ubuntu:18.04
```
3. Create the **self-define-images** folder, and edit **Dockerfile** and **test_app.py** in the folder for the custom image. In the sample code, the application code runs on the Flask framework.

The file structure is as follows:


```
self-define-images/
--Dockerfile
--test_app.py
```

– **Dockerfile**

```
From ubuntu:18.04
# Configure the source and install Python, Python3-PIP, and Flask.
RUN cp -a /etc/apt/sources.list /etc/apt/sources.list.bak && \
  sed -i "s@http://.*security.ubuntu.com@http://repo.xxx.com@g" /etc/apt/sources.list && \
  sed -i "s@http://.*archive.ubuntu.com@http://repo.xxx.com@g" /etc/apt/sources.list && \
  apt-get update && \
  apt-get install -y python3 python3-pip && \
  pip3 install --trusted-host https://repo.xxx.com -i https://repo.xxx.com/repository/pypi/simple
Flask

# Copy the application code to the image.
COPY test_app.py /opt/test_app.py

# Specify the boot command of the image.
CMD python3 /opt/test_app.py
```

– **test_app.py**

```
from flask import Flask, request
import json
app = Flask(__name__)

@app.route('/greet', methods=['POST'])
def say_hello_func():
    print("----- in hello func -----")
    data = json.loads(request.get_data(as_text=True))
    print(data)
    username = data['name']
    rsp_msg = 'Hello, {}'.format(username)
    return json.dumps({"response":rsp_msg}, indent=4)

@app.route('/goodbye', methods=['GET'])
def say_goodbye_func():
    print("----- in goodbye func -----")
    return '\nGoodbye!\n'

@app.route('/', methods=['POST'])
def default_func():
    print("----- in default func -----")
    data = json.loads(request.get_data(as_text=True))
    return '\n called default func !\n {}'.format(str(data))

# host must be "0.0.0.0", port must be 8080
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8080)
```

 **NOTE**

ModelArts forwards requests to port 8080 of the service started from the custom image. Therefore, the service listening port in the container must be port 8080. See the **test_app.py** file.

4. Switch to the **self-define-images** folder and run the following command to create custom image **test:v1**:

```
docker build -t test:v1 .
```
5. Run **docker image** to view the custom image you have created.

Verifying the Image Locally and Uploading It to SWR

1. Run the following command in the local environment to start the custom image:

```
docker run -it -p 8080:8080 test:v1
```

Figure 3-1 Starting a custom image

```
:/opt/file# docker run -it -p 8080:8080 test:v1
* Serving Flask app "test_app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

2. Open another terminal and run the following commands to test the functions of the three APIs of the custom image:

```
curl -X POST -H "Content-Type: application/json" --data '{"name":"Tom"}' 127.0.0.1:8080/
curl -X POST -H "Content-Type: application/json" --data '{"name":"Tom"}' 127.0.0.1:8080/greet
curl -X GET 127.0.0.1:8080/goodbye
```

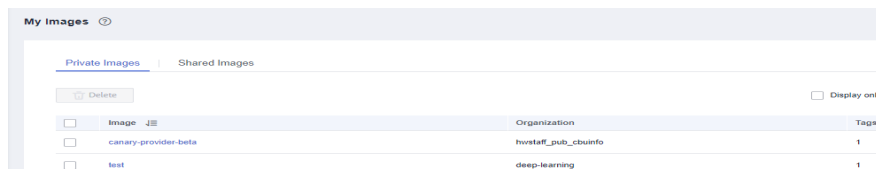
If information similar to the following is displayed, the function verification is successful.

Figure 3-2 Testing API functions

```
root@: ~# curl -X POST -H "Content-Type: application/json" --data '{"name":"Tom"}' 127.0.0.1:8080/
called default func !
{"name": "Tom"}
root@: ~# curl -X POST -H "Content-Type: application/json" --data '{"name":"Tom"}' 127.0.0.1:8080/greet
{"response": "Hello, Tom!"}
root@: ~# curl -X GET 127.0.0.1:8080/goodbye
Goodbye!
```

3. Upload the custom image to SWR. For details, see [How Can I Upload Images to SWR?](#)
4. View the uploaded image on the **My Images > Private Images** page of the SWR console.

Figure 3-3 Uploaded images

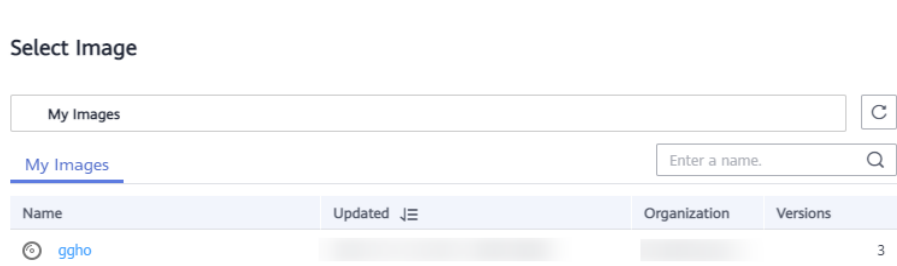


Using the Custom Image to Create an AI Application

Import a meta model. For details, see [Creating and Importing a Model Image](#). Key parameters are as follows:

- **Meta Model Source:** Select **Container image**.
 - **Container Image Path:** Select the created private image.

Figure 3-4 Created private image



- **Container API:** Protocol and port number for starting a model. This parameter is optional.
- **Health Check:** checks health status of a model. This parameter is optional. This parameter is configurable only when the health check API is configured in the custom image. Otherwise, creating the AI application will fail.
- **APIs:** APIs of a custom image. This parameter is optional. The model APIs must comply with ModelArts specifications. For details, see [Specifications for Compiling the Model Configuration File](#).

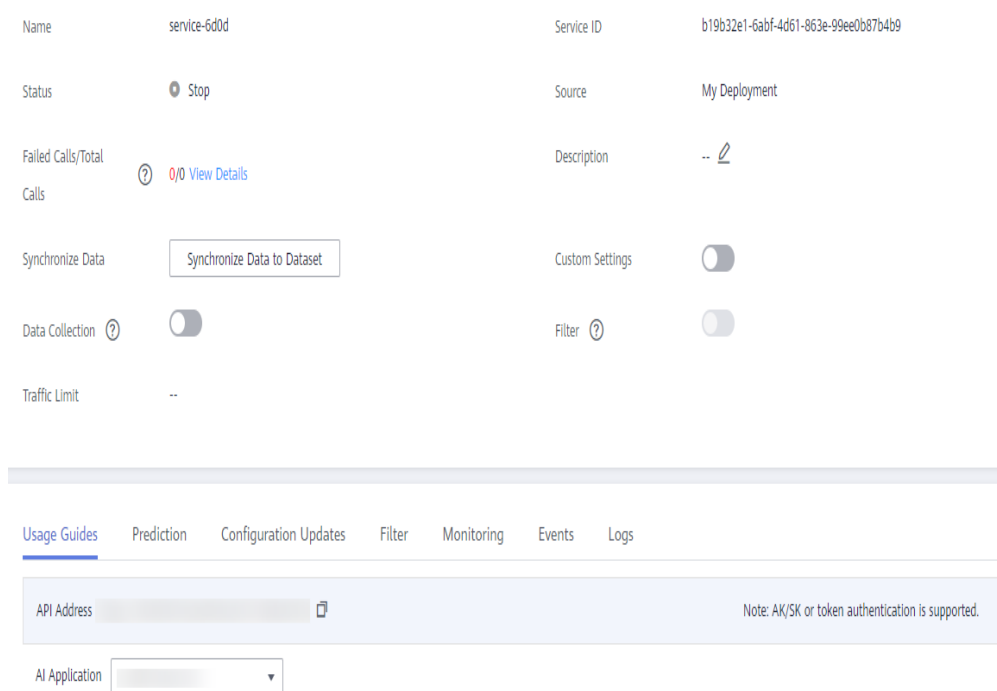
The configuration file is as follows:

```
[{
  "url": "/",
  "method": "post",
  "request": {
    "Content-type": "application/json"
  },
  "response": {
    "Content-type": "application/json"
  }
},
{
  "url": "/greet",
  "method": "post",
  "request": {
    "Content-type": "application/json"
  },
  "response": {
    "Content-type": "application/json"
  }
},
{
  "url": "/goodbye",
  "method": "get",
  "request": {
    "Content-type": "application/json"
  },
  "response": {
    "Content-type": "application/json"
  }
}
]
```

Deploying the AI Application as a Real-Time Service

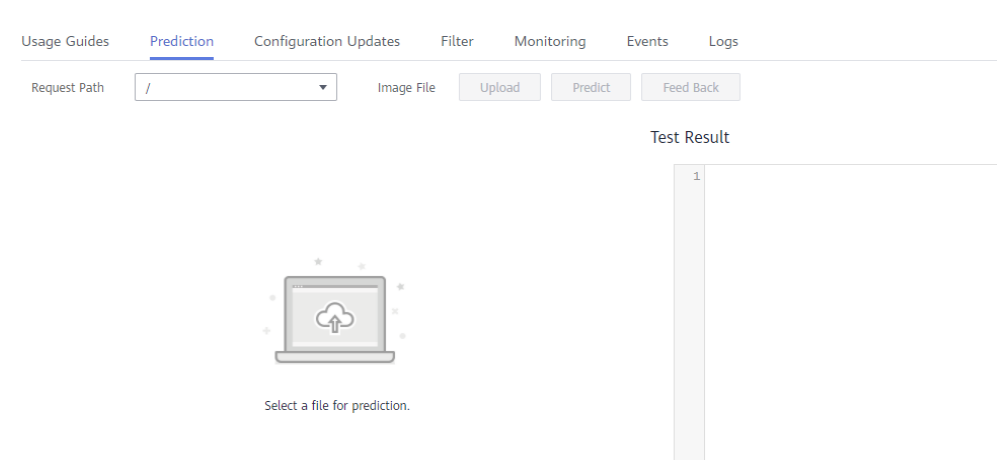
1. Deploy the AI application as a real-time service. For details, see [Deploying as a Real-Time Service](#).
2. View the details about the real-time service.

Figure 3-5 Usage Guides



3. Access the real-time service on the **Prediction** tab page.

Figure 3-6 Accessing a real-time service



3.2 End-to-End O&M of Inference Services

The end-to-end O&M of ModelArts inference services involves the entire AI process including algorithm development, service O&M, and service running.

Overview

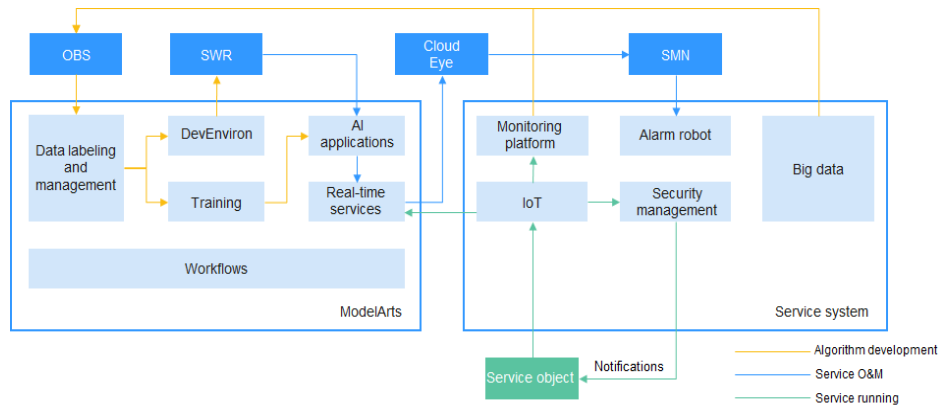
End-to-End O&M Process

- During algorithm development, store service data in Object Storage Service (OBS), and then label and manage the data using ModelArts data

management. After the data is trained, obtain an AI model and create AI application images using a development environment.

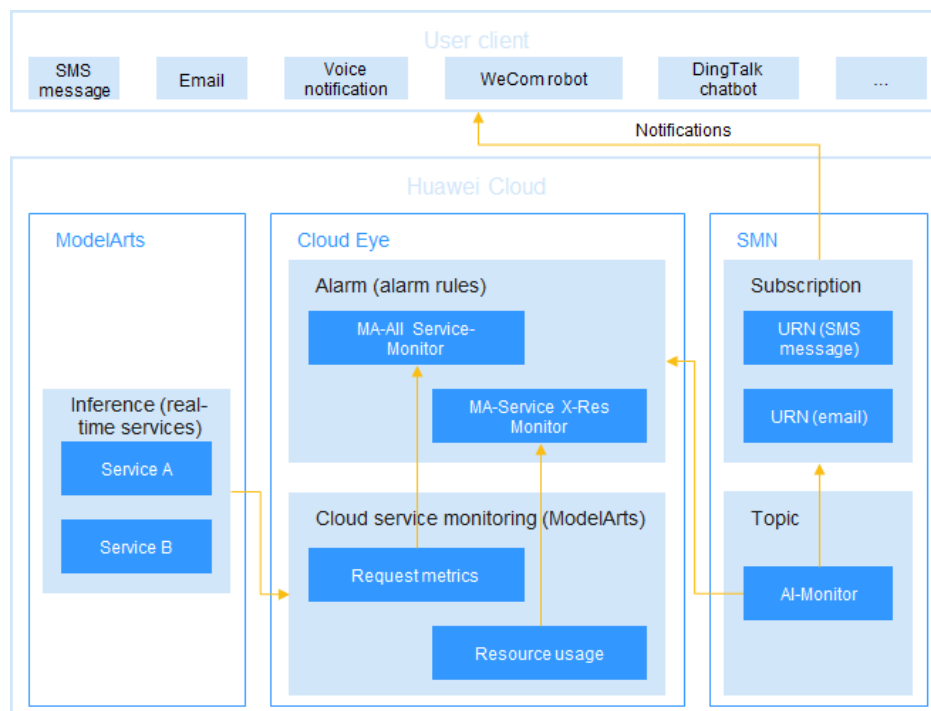
- During service O&M, use an image to create an AI application and deploy the AI application as a real-time service. You can obtain the monitoring data of the ModelArts real-time service on the Cloud Eye management console. Configure alarm rules so that you can be notified of alarms in real time.
- During service running, access real-time service requests into the service system and then configure service logic and monitoring.

Figure 3-7 End-to-end O&M process for inference services



During the entire O&M process, service request failures and high resource usage are monitored. When the resource usage threshold is reached, the system will send an alarm notification to you.

Figure 3-8 Alarming process



Advantages

End-to-end service O&M enables you to easily check service running at both peak and off-peak hours and detect the health status of real-time services in real time.

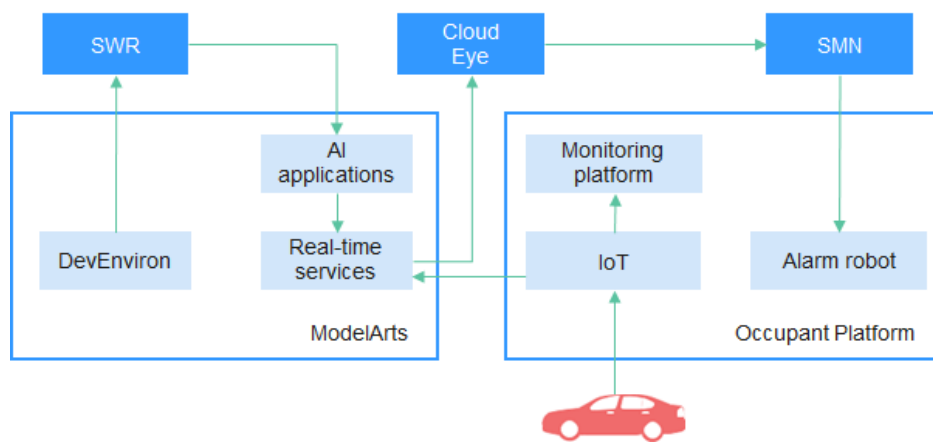
Constraints

End-to-end service O&M applies only to real-time services because Cloud Eye does not monitor batch or edge inference services.

Procedure

This section uses an occupant safety algorithm in travel as an example to describe how to use ModelArts for process-based service deployment and update, as well as automatic service O&M and monitoring.

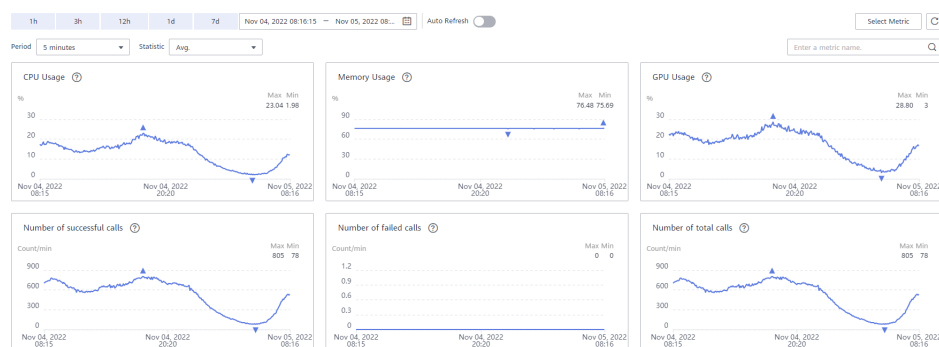
Figure 3-9 Occupant safety algorithm implementation



- Step 1** Use a locally developed model to create a custom image and use the image to create an AI application on ModelArts. For details, see .
- Step 2** On the ModelArts management console, deploy the created AI application as a real-time service.
- Step 3** Log in to the Cloud Eye management console, configure ModelArts alarm rules and enable notifications with a topic subscribed to. For details, see .

After the configuration, choose **Cloud Service Monitoring > ModelArts** in the navigation pane on the left to view the requests and resource usage of the real-time service.

Figure 3-10 Viewing service monitoring metrics



When an alarm is triggered based on the monitored data, the object who has subscribed to the target topic will receive a message notification.

----End

3.3 Creating an AI Application Using a Custom Engine

When you use a custom engine to create an AI application, you can select your image stored in SWR as the engine and specify a file directory in OBS as the model package. In this way, bring-your-own images can be used to meet your dedicated requirements.

Before deploying such an AI application as a service, ModelArts downloads the SWR image to the cluster and starts the image as a container as the user whose UID is 1000 and GID is 100. Then, ModelArts downloads the OBS file to the `/home/mind/model` directory in the container and runs the boot command preset in the SWR image. The service available to port 8080 in the container is automatically registered with APIG. You can access the service through the APIG URL.

Specifications for Using a Custom Engine to Create an AI Application

To use a custom engine to create an AI application, ensure the SWR image, OBS model package, and file size comply with the following requirements:

- SWR image specifications
 - A common user named **ma-user** in group **ma-group** must be built in the SWR image. Additionally, the UID and GID of the user must be 1000 and 100, respectively. The following is the dockerfile command for the built-in user:

```
groupadd -g 100 ma-group && useradd -d /home/ma-user -m -u 1000 -g 100 -s /bin/bash ma-user
```
 - Specify a command for starting the image. In the dockerfile, specify **cmd**. The following shows an example:

```
CMD sh /home/mind/run.sh
```

Customize the startup entry file **run.sh**. The following is an example.

```
#!/bin/bash

# User-defined script content
...

# run.sh calls app.py to start the server. For details about app.py, see "HTTPS Example".
python app.py
```
 - The service must be HTTPS enabled, and it is available on port 8080. For details, see the [HTTPS example](#).
 - (Optional) On port 8080, enable health check with URL **/health**. (The health check URL must be **/health**.)
- OBS model package specifications
 - The name of the model package must be **model**. For details about model package specifications, see [Introduction to Model Package Specifications](#).
- File size specifications

When a public resource pool is used, the total size of the downloaded SWR image (not the compressed image displayed on the SWR page) and the OBS model package cannot exceed 30 GB.

HTTPS Example

Use Flask to start HTTPS. The following is an example of the web server code:

```
from flask import Flask, request
import json

app = Flask(__name__)

@app.route('/greet', methods=['POST'])
def say_hello_func():
    print("----- in hello func -----")
    data = json.loads(request.get_data(as_text=True))
    print(data)
    username = data['name']
    rsp_msg = 'Hello, {}'.format(username)
    return json.dumps({"response":rsp_msg}, indent=4)

@app.route('/goodbye', methods=['GET'])
def say_goodbye_func():
    print("----- in goodbye func -----")
    return '\nGoodbye!\n'

@app.route('/', methods=['POST'])
def default_func():
    print("----- in default func -----")
    data = json.loads(request.get_data(as_text=True))
    return '\n called default func !\n {} \n'.format(str(data))

@app.route('/health', methods=['GET'])
def healthy():
    return "{\"status\": \"OK\"}"

# host must be "0.0.0.0", port must be 8080
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8080, ssl_context='adhoc')
```

Debugging on a Local Computer

Perform the following operations on a local computer with Docker installed to check whether a custom engine complies with specifications:

1. Download the custom image, for example, **custom_engine:v1** to the local computer.
2. Copy the model package folder **model** to the local computer.
3. Run the following command in the same directory as the model package folder to start the service:

```
docker run --user 1000:100 -p 8080:8080 -v model:/home/mind/model custom_engine:v1
```

NOTE

This command is used for simulation only because the directory mounted to **-v** is assigned the root permission. In the cloud environment, after the model file is downloaded from OBS to **/home/mind/model**, the file owner will be changed to **ma-user**.

4. Start another terminal on the local computer and run the following command to obtain the expected inference result:


```
curl https://127.0.0.1:8080/${Request path to the inference service}
```

Deployment Example

The following section describes how to use a custom engine to create an AI application.

1. Create an AI application and viewing its details.

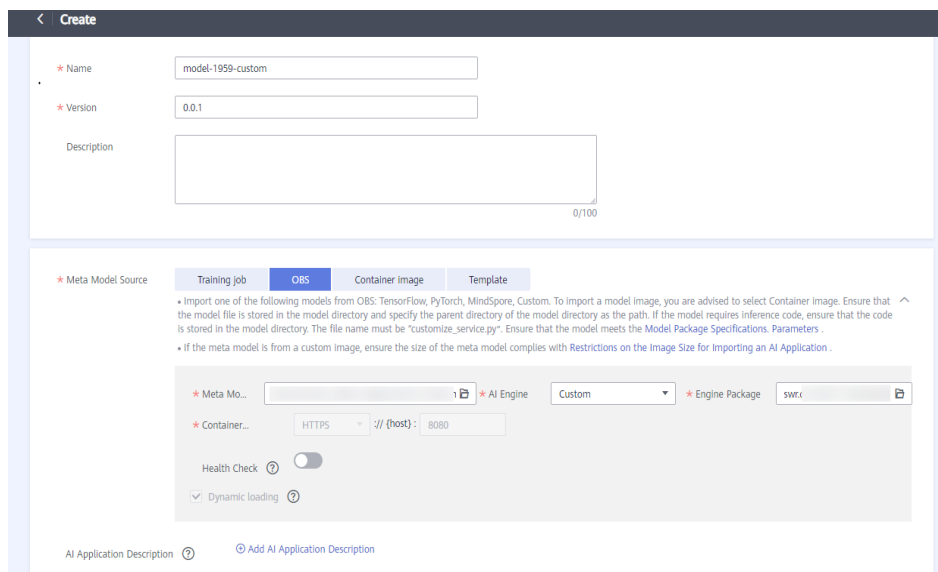
Log in to the ModelArts console, choose **AI Application Management > AI Applications**, and click **Create**. On the page that is displayed, configure the following parameters:

- **Meta Model Source:** OBS
- **Meta Model:** a model package selected from OBS
- **AI Engine:** Custom
- **Engine Package:** an SWR image

Retain the default settings for other parameters.

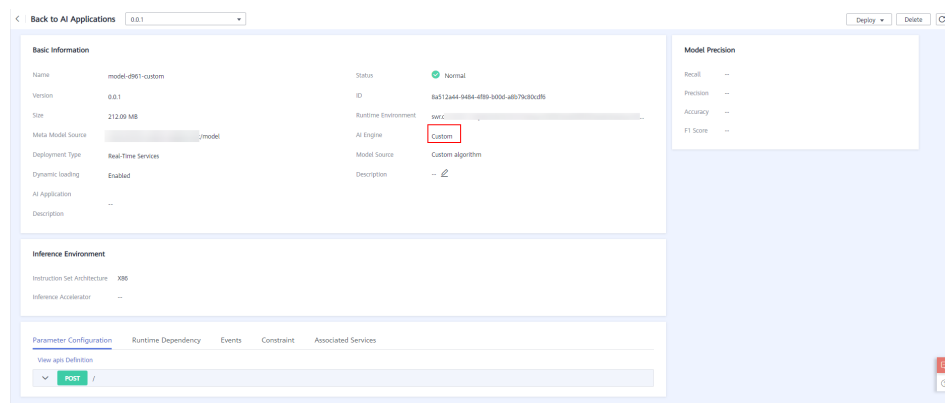
Click **Create Now**. In the AI application list that is displayed, check the AI application status. When its status changes to **Normal**, the AI application has been created.

Figure 3-11 Creating an AI application



Click the AI application name. On the page that is displayed, view details about the AI application.

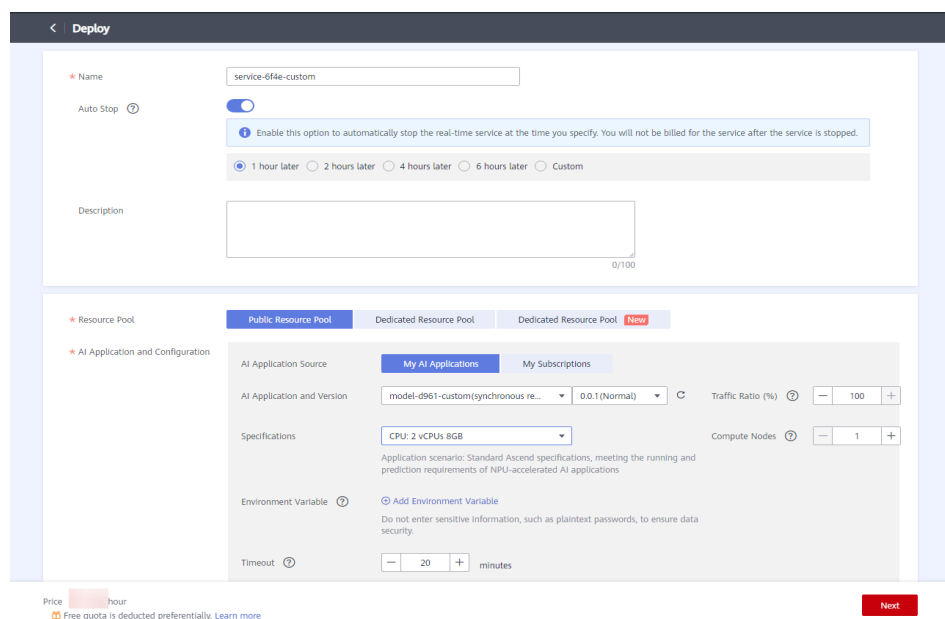
Figure 3-12 Viewing details about an AI application



2. Deploy the AI application as a service and view service details.

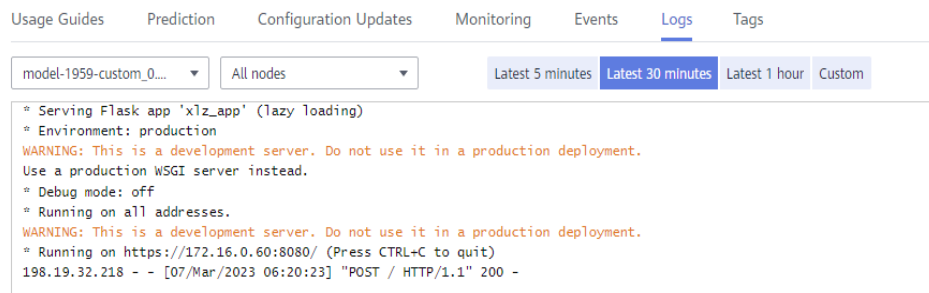
On the AI application details page, choose **Deploy** > **Real-Time Services** in the upper right corner. On the **Deploy** page, select a proper compute node specification, retain the default settings for other parameters, and click **Next**. When the service status changes to **Running**, the service has been deployed.

Figure 3-13 Deploying a service



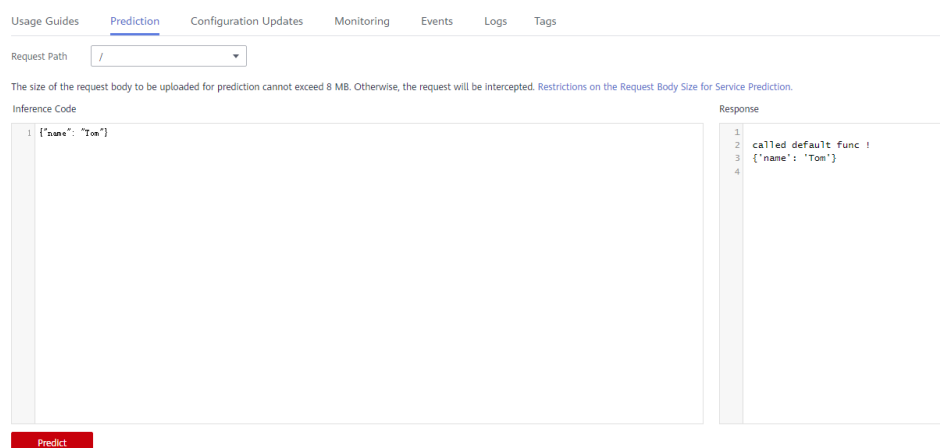
Click the service name. On the page that is displayed, view the service details. Click the **Logs** tab to view the service logs.

Figure 3-14 Logs



- Use the service for prediction.
On the service details page, click the **Prediction** tab to use the service for prediction.

Figure 3-15 Prediction



3.4 Using a Large Model to Create an AI Application and Deploying a Real-Time Service

Context

Currently, a large model can have hundreds of billions or even trillions of parameters, and its size becomes larger and larger. A large model with hundreds of billions of parameters exceeds 200 GB, and poses new requirements for version management and production deployment of the platform. For example, importing AI applications requires dynamic adjustment of the tenant storage quota. Slow model loading and startup requires a flexible timeout configuration in the deployment. The service recovery time needs to be shortened in the event that the model needs to be reloaded upon a restart caused by a load exception.

To address the preceding requirements, the ModelArts inference platform provides a solution to AI application management and service deployment in large model application scenarios.

Constraints

- You need to apply for the size quota of an AI application and add the whitelist cached using the local storage of the node.
- You need to use the custom engine **Custom** to configure dynamic loading.
- A dedicated resource pool is required to deploy the service.
- The disk space of the dedicated resource pool must be greater than 1 TB.

Procedure

1. [Applying for Increasing the Size Quota of an AI Application and Using the Local Storage of the Node to Cache the Whitelist](#)
2. [Uploading Model Data and Verifying the Consistency of Uploaded Objects](#)
3. [Creating a Dedicated Resource Pool](#)
4. [Creating an AI Application](#)
5. [Deploying a Real-Time Service](#)

Applying for Increasing the Size Quota of an AI Application and Using the Local Storage of the Node to Cache the Whitelist

During service deployment, the dynamically loaded model package is stored in the temporary disk space by default. When the service is stopped, the loaded files are deleted, and they need to be reloaded when the service is restarted. To avoid repeated loading, the platform allows the model package to be loaded from the local storage space of the node in the resource pool and keeps the loaded files valid even when the service is stopped or restarted (using the hash value to ensure data consistency).

To use a large model, you need to use a custom engine and enable dynamic loading when importing the model. In this regard, you need to perform the following operations:

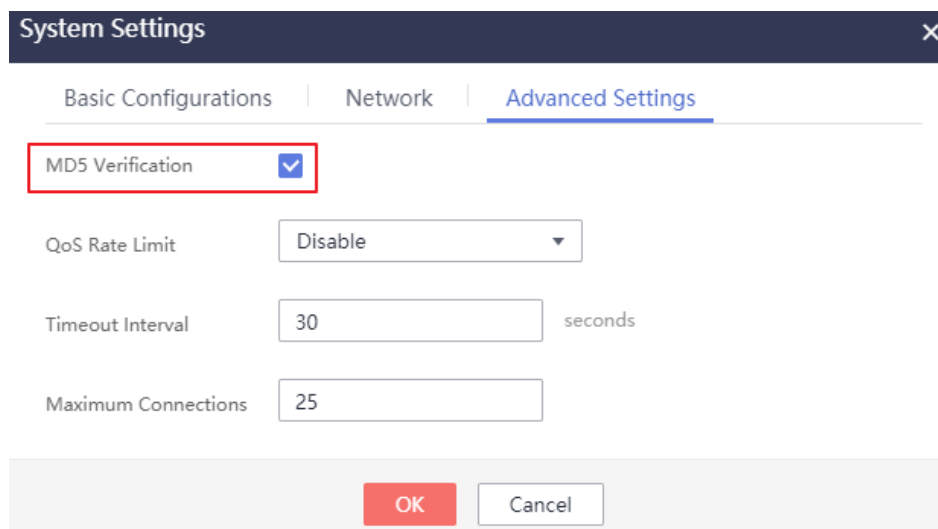
- If the model size exceeds the default quota, submit a service ticket to increase the size quota of a single AI application. The default size quota of an AI application is 20 GB.
- Submit a service ticket to add the whitelist cached using the local storage of the node.

Uploading Model Data and Verifying the Consistency of Uploaded Objects

To ensure data integrity during dynamic loading, you need to verify the consistency of uploaded objects when uploading model data to OBS. obsutil, OBS Browser+, and OBS SDKs support verification of data consistency during upload. You can select a method that meets your requirements. For details, see "Verifying Data Consistency During Upload" in Object Storage Service documentation.

For example, if you upload data via OBS Browser+, enable MD5 verification, as shown in [Figure 3-16](#). When dynamic loading is enabled and the local persistent storage of the node is used, OBS Browser+ checks data consistency during data upload.

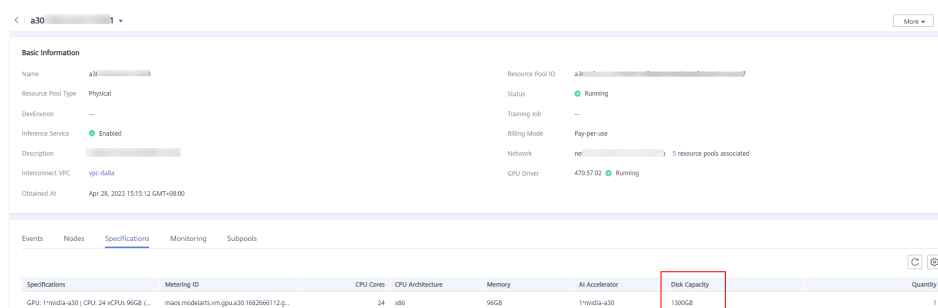
Figure 3-16 Configuring MD5 verification for OBS Browser+



Creating a Dedicated Resource Pool

To use the local persistent storage, you need to create a dedicated resource pool whose disk space is greater than 1 TB. You can view the disk information on the **Specifications** tab of the **Basic Information** page of the dedicated resource pool. If a service fails to be deployed and the system displays a message indicating that the disk space is insufficient, see [What Do I Do If Resources Are Insufficient When a Real-Time Service Is Deployed, Started, Upgraded, or Modified.](#)

Figure 3-17 Viewing the disk information of the dedicated resource pool



Creating an AI Application

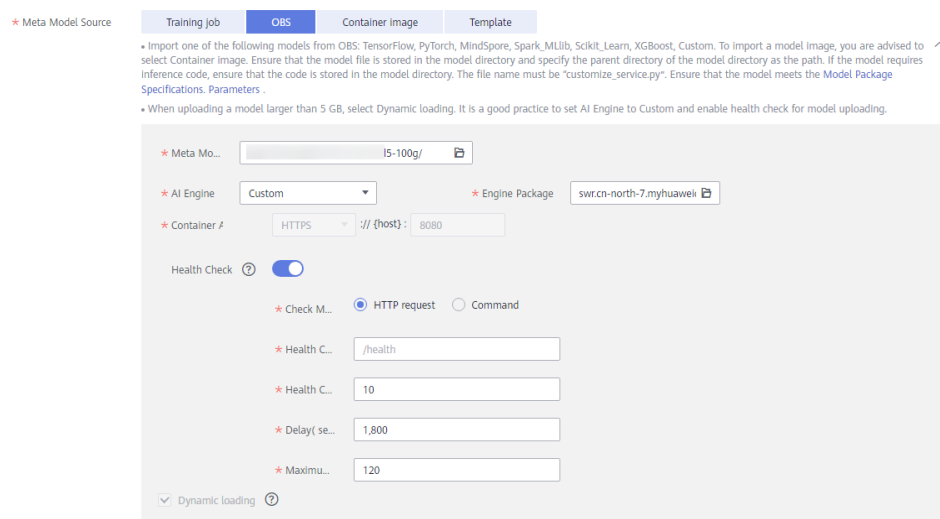
If you use a large model to create an AI application and import the model from OBS, complete the following configurations:

1. Use a custom engine and enable dynamic loading.
To use a large model, you need to use a custom engine and enable dynamic loading when importing the model. You can create a custom engine to meet special requirements for image dependency packages and inference frameworks in large model scenarios. For details about how to create a custom engine, see [Creating an AI Application Using a Custom Engine.](#)
When you use a custom engine, dynamic loading is enabled by default. The model package is separated from the image, and the model is dynamically loaded to the service load during service deployment.

2. Configure health check.

Health check is mandatory for the AI applications imported using a large model to identify unavailable services that are displayed as started.

Figure 3-18 Using a custom engine, enabling dynamic loading, and configuring health check



Deploying a Real-Time Service

When deploying the service, complete the following configurations:

1. Customize the deployment timeout interval.

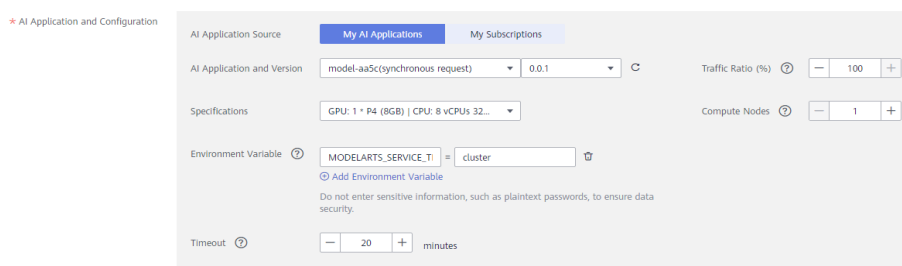
Generally, the time for loading and starting a large model is longer than that for a common model. Set **Timeout** to a proper value. Otherwise, the timeout may elapse prior to the completion of the model startup, and the deployment may fail.

2. Add an environment variable.

During service deployment, add the following environment variable to set the service traffic load balancing policy to cluster affinity, preventing unready service instances from affecting the prediction success rate:

```
MODELARTS_SERVICE_TRAFFIC_POLICY: cluster
```

Figure 3-19 Customizing the deployment timeout interval and adding an environment variable



You are advised to deploy multiple instances to improve service reliability.

3.5 High-Speed Access to Inference Services Through VPC Peering

Context

When accessing a real-time service, you may require:

- High throughput and low latency
- TCP or RPC requests

To meet these requirements, ModelArts enables high-speed access through VPC peering.

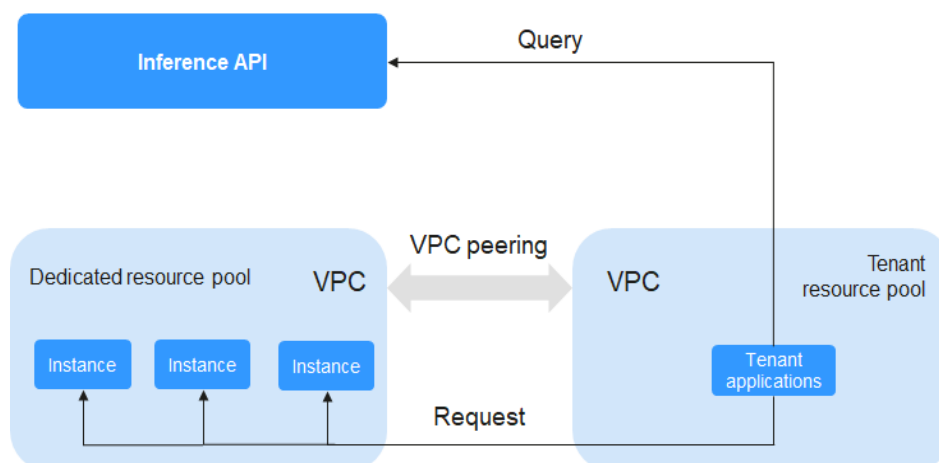
In high-speed access through VPC peering, your service requests are directly sent to instances through VPC peering but not through the inference platform. This accelerates service access.

NOTE

The following functions that are available through the inference platform will be unavailable if you use high-speed access:

- Authentication
- Traffic distribution by configuration
- Load balancing
- Alarm, monitoring, and statistics

Figure 3-20 High-speed access through VPC peering



Preparations

Deploy a real-time service in a dedicated resource pool and ensure the service is running.

NOTICE

- For details about how to deploy services in new-version dedicated resource pools, see [Comprehensive Upgrades to ModelArts Resource Pool Management Functions](#).
- Only the services deployed in a dedicated resource pool support high-speed access through VPC peering.
- High-speed access through VPC peering is available only for real-time services.
- Due to traffic control, the number of calls of each tenant account cannot exceed 2000 per minute, and that of each IAM user account cannot exceed 20 per minute.
- High-speed access through VPC peering is available only for the services deployed using the AI applications imported from custom images.

Procedure

To enable high-speed access to a real-time service through VPC peering, perform the following operations:

1. [Interconnect the dedicated resource pool to the VPC.](#)
2. [Create an ECS in the VPC.](#)
3. [Obtain the IP address and port number of the service.](#)
4. [Access the service through the IP address and port number.](#)

Step 1 Interconnect the dedicated resource pool to the VPC.

Log in to the ModelArts management console, choose **Dedicated Resource Pools** > **Elastic Cluster**, locate the dedicated resource pool used for service deployment, and click its name/ID to go to the resource pool details page. Obtain the network configuration. Switch back to the dedicated resource pool list, click the **Network** tab, locate the network associated with the dedicated resource pool, and interconnect it with the VPC. After the VPC is accessed, the VPC will be displayed on the network list and resource pool details pages. Click the VPC to go to the details page.

Figure 3-21 Locating the target dedicated resource pool

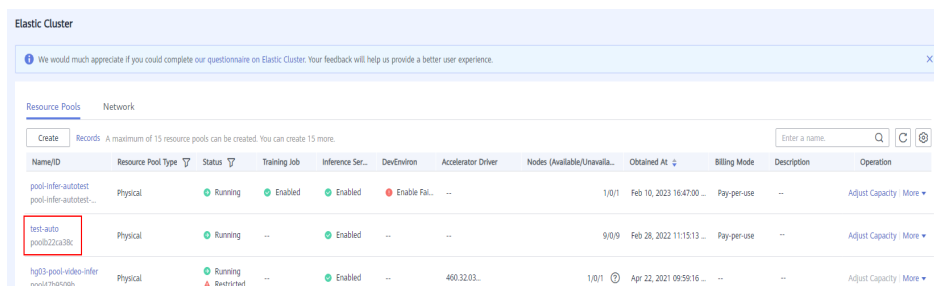


Figure 3-22 Obtaining the network configuration

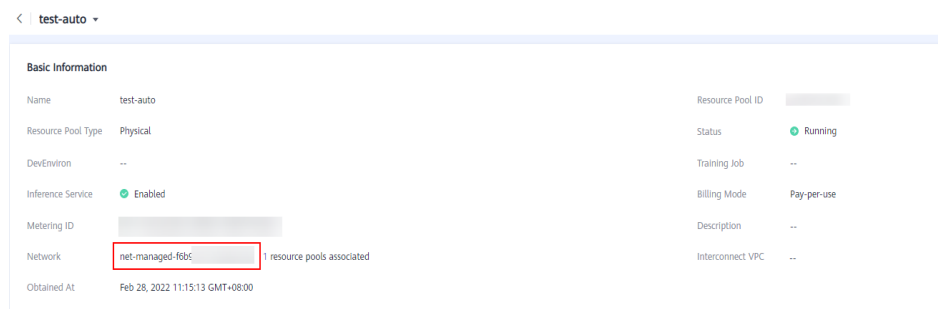
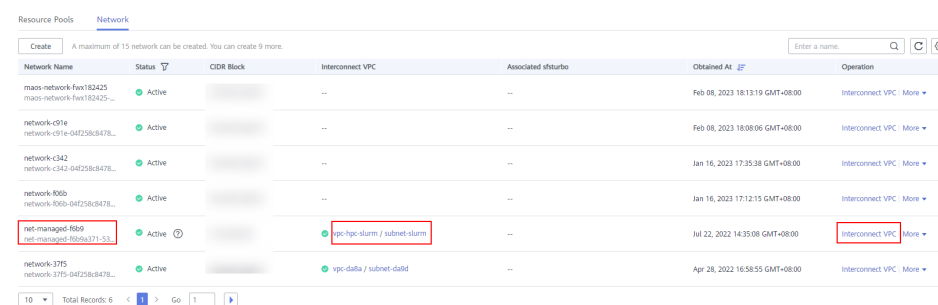


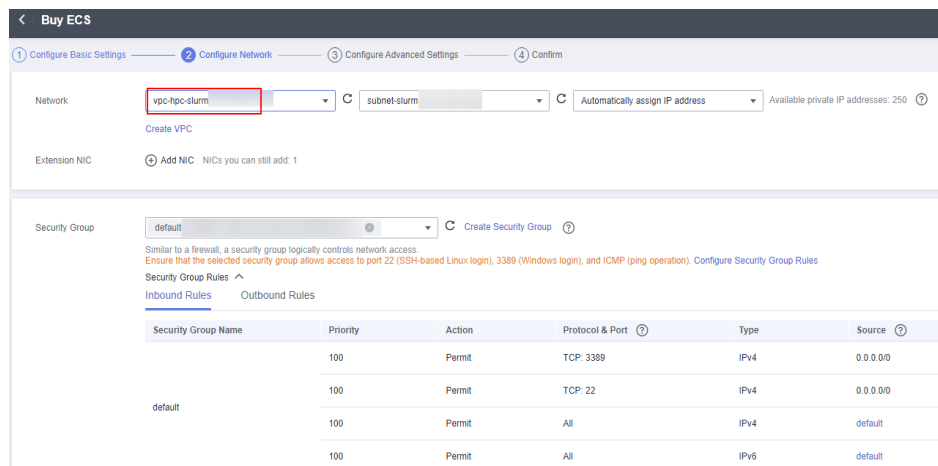
Figure 3-23 Interconnecting the VPC



Step 2 Create an ECS in the VPC.

Log in to the ECS management console and click **Buy ECS** in the upper right corner. On the **Buy ECS** page, configure basic settings and click **Next: Configure Network**. On the **Configure Network** page, select the VPC connected in **Step 1**, configure other parameters, confirm the settings, and click **Submit**. When the ECS status changes to **Running**, the ECS has been created. Click its name/ID to go to the server details page and view the VPC configuration.

Figure 3-24 Purchasing an ECS



ECS Information

ID	[REDACTED]
Name	ecs-zxy
Region	North-Ulanqab203
AZ	AZ1
Specifications	General computing 2 vCPUs 16 GiB m2.large.8
Image	CentOS 8.0 64bit for Tenant 20210227 Public image
VPC	vpc-hpc-slurm

Billing Mode	Yearly/Monthly
Order	[REDACTED]
Obtained	Mar 02, 2023 16:40:41 GMT+08:00
Launched	Mar 02, 2023 16:40:56 GMT+08:00
Expires On	Apr 02, 2023 23:59:59 GMT+08:00

Step 3 Obtain the IP address and port number of the service.

GUI software, for example, Postman can be used to obtain the IP address and port number. Alternatively, log in to the ECS, create a Python environment, and execute code to obtain the service IP address and port number.

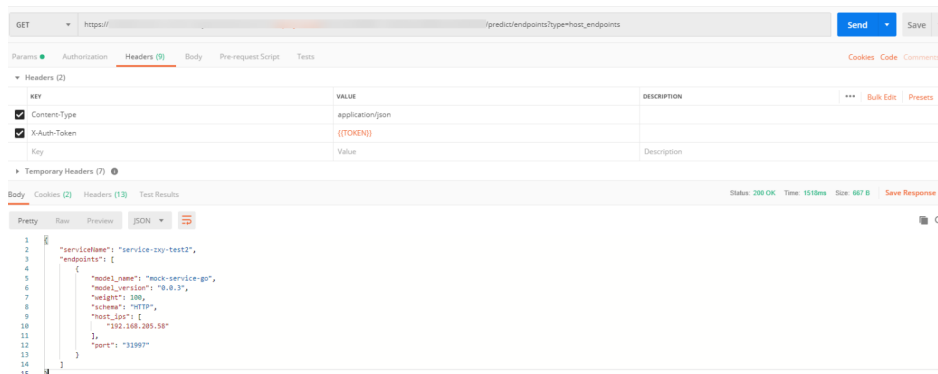
API:

```
GET /v1/{project_id}/services/{service_id}/predict/endpoints?type=host_endpoints
```

For details about how to obtain a service endpoint, see "Before You Start" > "Endpoints" in *ModelArts API Reference*.

- Obtain the IP address and port number using GUI software.

Figure 3-25 Example response



- Obtain the IP address and port number using Python.
The Python code is as follows (mandatory parameters must be configured):

```
def get_app_info(project_id, service_id):  
    list_host_endpoints_url = "{}/v1/{}/services/{}/predict/endpoints?type=host_endpoints"  
    url = list_host_endpoints_url.format(REGION_ENDPOINT, project_id, service_id)  
    headers = {'X-Auth-Token': X_Auth-Token}  
    response = requests.get(url, headers=headers)  
    print(response.content)
```

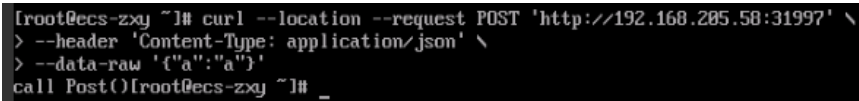
Step 4 Access the service through the IP address and port number.

Log in to the ECS and access the real-time service either by running Linux commands or by creating a Python environment and executing Python code. Obtain the values of **schema**, **ip**, and **port** from [Step 3](#).

- Run the following command to access the real-time service:

```
curl --location --request POST 'http://192.168.205.58:31997' \  
--header 'Content-Type: application/json' \  
--data-raw '{"a":"a"}'
```

Figure 3-26 Accessing a real-time service



```
[root@ecs-zxy ~]# curl --location --request POST 'http://192.168.205.58:31997' \  
> --header 'Content-Type: application/json' \  
> --data-raw '{"a":"a"}'  
call Post()[root@ecs-zxy ~]# _
```

- Create a Python environment and execute Python code to access the real-time service.

```
def vpc_infer(schema, ip, port, body):  
    infer_url = "{}/{}:{}".format(schema, ip, port)  
    url = infer_url.format(schema, ip, port)  
    response = requests.post(url, data=body)  
    print(response.content)
```

----End