ModelArts

Inference Deployment

Issue   01
Date   2025-01-06

Contents

1 Introduction to Inference

After an AI model is developed, you can use it to create an AI application and quickly deploy the application as an inference service. The AI inference capabilities can be integrated into your IT platform by calling APIs.

Figure 1-1 Inference
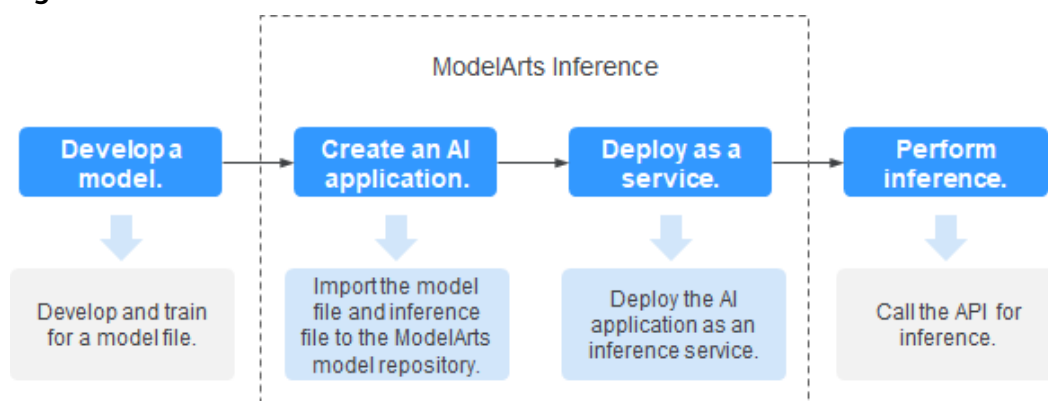
- Develop a model: Models can be developed in ModelArts or your local development environment. A locally developed model must be uploaded to Huawei Cloud OBS.
- Create an AI application: Import the model file and inference file to the ModelArts model repository and manage them by version. Use these files to build an executable AI application.
- Deploy as a service: Deploy the AI application as a container instance in the resource pool and register inference APIs that can be accessed externally.
- Perform inference: Add the function of calling the inference APIs to your application to integrate AI inference into the service process.

Deploying an AI Application as a Service

After an AI application is created, you can deploy it as a service on the **Deploy** page. ModelArts supports the following deployment types:

- **Real-time service**

  Deploy an AI application as a web service with real-time test UI and monitoring supported.

- **Batch service**

  Deploy an AI application as a batch service that performs inference on batch data and automatically stops after data processing is complete.

2 Managing AI Applications

2.1 Introduction to AI Application Management

AI development and optimization require frequent iterations and debugging. Modifications in datasets, training code, or parameters affect the quality of models. If the metadata of the development process cannot be centrally managed, the optimal model may fail to be reproduced.

ModelArts imports all meta models obtained through training. You can also create AI applications using meta models from OBS or container images. In this way, you can centrally manage all iterated and debugged AI applications.

Constraints

- In an ExeML project, after a model is deployed, the model is automatically uploaded to the AI application list. However, AI applications generated by ExeML cannot be downloaded and can be used only for deployment and rollout.
- AI application creation and AI application version management are available for free to all users.

Scenarios for Creating AI Applications

- **Imported from a training job**: Create a training job in ModelArts and train a model. After obtaining a satisfactory model, use it to create an AI application and deploy the application as services.
- **Imported from OBS**: If you use a mainstream framework to develop and train a model locally, you can upload the model to an OBS bucket based on the

model package specifications, import the model from OBS to ModelArts, and use the model to create an AI application for service deployment.

- **Imported from a container image**: If an AI engine is not supported by ModelArts, you can use it to build a model, import the model to ModelArts as a custom image, use the image to create an AI application, and deploy the AI application as services.

- **Imported from a template**: Since the configurations of models with the same functions are similar, ModelArts integrates the configurations of such models into a general template. By using this template, you can easily and quickly import models and create AI applications without writing the **config.json** configuration file.

☐ **NOTE**

> Importing a model from a template will be unavailable soon. After it goes offline, you can use the templates for AI engine and model configurations by choosing **OBS**, setting **AI Engine** to **Custom**, and importing your custom AI engine.

AI Application-related Functions

Table 2-1 AI application-related functions

Supported Function	Description
Creating an AI Application	Import the trained models to ModelArts and create AI applications for centralized management. The following provides the operation guide for each method of importing models. • **Importing a Meta Model from a Training Job** • **Importing a Meta Model from OBS** • **Importing a Meta Model from a Container Image** • **Importing a Meta Model from a Template**
Viewing Details About an AI Application	After an AI application is created, you can view its information on the details page.
Managing AI Application Versions	To facilitate traceback and model tuning, ModelArts provides the AI application version management function. You can manage AI applications by version.

Supported AI Engines for ModelArts Inference

If you import a model from a template or OBS to create an AI application, the following AI engines and versions are supported.

> **NOTE**
>
> - Runtime environments marked with **recommended** are unified runtime images, which will be used as mainstream base inference images. The installation packages of unified images are richer. For details, see **Base Inference Images**.
> - Images of the old version will be discontinued. Use unified images.
> - The base images to be removed are no longer maintained.
> - Naming a unified runtime image: *<AI engine name and version>* - *<Hardware and version*: CPU, CUDA, or CANN> - *<Python version>* - *<OS version>* - *<CPU architecture>*

Table 2-2 Supported AI engines and their runtime

Engine	Runtime	Note
TensorFlow	python3.6 python2.7 (unavailable soon) tf1.13-python3.6-gpu tf1.13-python3.6-cpu tf1.13-python3.7-cpu tf1.13-python3.7-gpu tf2.1-python3.7 (unavailable soon) tensorflow_2.1.0-cuda_10.1-py_3.7-ubuntu_18.04-x86_64 (recommended)	- TensorFlow 1.8.0 is used in **python2.7** and **python3.6**. - **python3.6**, **python2.7**, and **tf2.1-python3.7** indicate that the model can run on both CPUs and GPUs. For other runtime values, if the suffix contains **cpu** or **gpu**, the model can run only on CPUs or GPUs. - The default runtime is **python2.7**.
Spark_MLlib	python2.7 (unavailable soon) python3.6 (unavailable soon)	- Spark_MLlib 2.3.2 is used in **python2.7** and **python3.6**. - The default runtime is **python2.7**. - **python2.7** and **python3.6** can only be used to run models on CPUs.
Scikit_Learn	python2.7 (unavailable soon) python3.6 (unavailable soon)	- Scikit_Learn 0.18.1 is used in **python2.7** and **python3.6**. - The default runtime is **python2.7**. - **python2.7** and **python3.6** can only be used to run models on CPUs.
XGBoost	python2.7 (unavailable soon) python3.6 (unavailable soon)	- XGBoost 0.80 is used in **python2.7** and **python3.6**. - The default runtime is **python2.7**. - **python2.7** and **python3.6** can only be used to run models on CPUs.

Engine	Runtime	Note
PyTorch	python2.7 (unavailable soon) python3.6 python3.7 pytorch1.4-python3.7 pytorch1.5-python3.7 (unavailable soon) pytorch_1.8.0-cuda_10.2-py_3.7-ubuntu_18.04-x86_64 (recommended)	- PyTorch 1.0 is used in **python2.7**, **python3.6**, and **python3.7**. **python2.7**, **python3.6**, **python3.7**, **pytorch1.4-python3.7**, and **pytorch1.5-python3.7** indicate that the model can run on both CPUs and GPUs. - The default runtime is **python2.7**.
MindSpore	aarch64 (recommended)	AArch64 can run only on Snt3 chips.

2.2 Creating an AI Application

2.2.1 Importing a Meta Model from a Training Job

You can create a training job in ModelArts to obtain a satisfactory model. Then, you can import the model to **AI Application Management** for centralized management. In addition, you can quickly deploy the model as a service.

Constraints

- A model generated from a training job that uses a subscribed algorithm can be directly imported to ModelArts without the need to use the inference code or configuration file.
- If the meta model is from a container image, ensure the size of the meta model complies with **Restrictions on the Size of an Image for Importing an AI Application**.

Prerequisites

- The training job has been successfully executed, and the model has been stored in the OBS directory where the training output is stored (the input parameter is **train_url**).
- If a model is generated from a training job that uses a frequently-used framework or custom image, upload the inference code and configuration file to the storage directory of the model by referring to **Introduction to Model Package Specifications**.
- The OBS directory you use and ModelArts are in the same region.

Creating an AI Application

1. Log in to the ModelArts console, and choose **AI Applications** from the navigation pane. The **AI Applications** page is displayed.
2. Click **Create** in the upper left corner.
3. On the displayed page, set the parameters.

   a. Set basic information about the AI application. For details about the parameters, see **Table 2-3**.

   Table 2-3 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Version	Version of the AI application to be created. For the first import, the default value is **0.0.1**. **NOTE** After an AI application is created, you can **create new versions** using different meta models for optimization.
Description	Brief description of an AI application

   b. Select the meta model source and set related parameters. Set **Meta Model Source** to **Training job**. For details about the parameters, see **Table 2-4**.

   Figure 2-1 Setting a training job as the meta model source
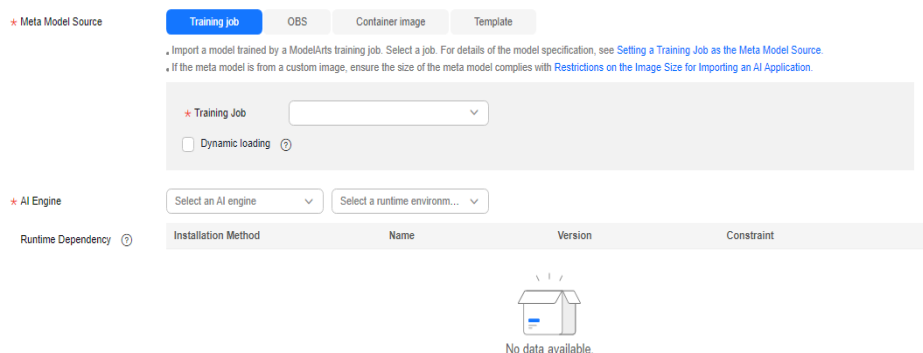
Table 2-4 Parameters of the meta model source

Parameter	Description
Meta Model Source	Choose **Training Job** > **Training Jobs** or **Training Job** > **Training Jobs (New)**. ● Choose a training job that has executed under the current account and a training version. ● **Dynamic loading**: enabled for quick deployment and model update. If this function is selected, model files and runtime dependencies are pulled only during an actual deployment. Enable this function if a single model file is larger than 5 GB. **NOTE** ModelArts provides model training of both the new and old versions. Training management of the old version is only available for its existing users.
AI Engine	Inference engine used by the meta model. The engine is automatically matched based on the training job you select.
Inference Code	Set inference code for an AI application. The code is used to customize the inference processing logic. Display the inference code URL. You can copy this URL directly.
Runtime Dependency	List the dependencies of the selected model in the environment. For example, if **tensorflow** is used and the installation method is **pip**, the version must be 1.8.0 or later.
AI Application Description	Provide AI application descriptions to help other AI application developers better understand and use your applications. Click **Add AI Application Description** and set the **Document name** and URL. A maximum of three AI application descriptions are supported.
Deployment Type	Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select **Real-time services** here, you can only deploy the AI application as a real-time service after it is created.

c. Confirm the configurations and click **Create now**. The AI application is created.

In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is successfully created. On this page, you can perform such operations as creating new versions and quickly deploying services.

Follow-Up Procedure

Deploying an AI Application as a Service: In the AI application list, click the option button on the left of the AI application name to display the version list at

the bottom of the list page. Locate the row that contains the target version, click **Deploy** in the **Operation** column to deploy the AI application as a service type selected during AI application creation.

2.2.2 Importing a Meta Model from a Template

 NOTE

> Importing a model from a template will be unavailable soon. After it goes offline, you can use the templates for AI engine and model configurations by choosing **OBS**, setting **AI Engine** to **Custom**, and importing your custom AI engine.

Because the configurations of models with the same functions are similar, ModelArts integrates the configurations of such models into a common template. By using this template, you can easily and quickly create AI applications without compiling the **config.json** configuration file. For details about the template, see **Introduction to Model Templates**.

Constraints

- For details about the supported templates, see **Supported Templates**. For details about the input and output modes of each template, see **Supported Input and Output Modes**.
- Creating and managing AI applications is free of charge.

Prerequisites

- Ensure that you have uploaded the model to OBS according to the model package specifications of the corresponding template.
- The OBS directory you use and ModelArts are in the same region.

Creating an AI Application

1. Log in to the ModelArts console, and choose **AI Applications** from the navigation pane. The **AI Applications** page is displayed.
2. Click **Create** in the upper left corner.
3. On the displayed page, set the parameters.

   a. Set basic information about the AI application. For details about the parameters, see **Table 2-5**.

Table 2-5 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Version	Version of the AI application to be created. For the first import, the default value is **0.0.1**. **NOTE** After an AI application is created, you can **create new versions** using different meta models for optimization.

Parameter	Description
Description	Brief description of an AI application

b. Select the meta model source and set related parameters. Set **Meta Model Source** to **Template**. For details about the parameters, see **Table 2-6**.

Table 2-6 Parameters of the meta model source

Parameter	Description
Model Template	Select a template from the existing ModelArts template list , such as **TensorFlow-based image classification template**. ModelArts also provides three filter criteria: **Type**, **Engine**, and **Environment**, helping you quickly find the desired template. If the three filter criteria cannot meet your requirements, you can enter keywords to search for the target template. For details about the supported templates, see **Supported Templates**.
Model Directory	OBS path where a model is saved. Select an OBS path for storing the model based on the input requirements of the selected model template. The OBS path cannot contain spaces. Otherwise, the AI application fails to be created. **NOTE** ● If you select an encrypted bucket or file, the import will fail. ● If a training job is executed for multiple times, different version directories are generated, such as V001 and V002, and the generated models are stored in the **model** folder in different version directories. When selecting model files, specify the **model** folder in the corresponding version directory.
Dynamic Loading	Quick deployment and model update. If this function is selected, model files and runtime dependencies are pulled only during an actual deployment. Select this function if the size of a single model file exceeds 5 GB.

Parameter	Description
Input and Output Mode	If the default input and output mode of the selected template can be overwritten, you can select an input and output mode based on the AI application function or application scenario. **Input and Output Mode** is an abstract of the API (**apis**) in **config.json**. It describes the interface provided by the AI application for external inference. An input and output mode describes one or more APIs, and corresponds to a template. For example, for **TensorFlow-based image classification template**, **Input and Output Mode** supports **Built-in image processing mode**. The input and output mode cannot be modified in the template. Therefore, you can only view but not modify the default input and output mode of the template on the page. For details about the supported input and output modes, see **Supported Input and Output Modes**.
AI Application Description	Provide AI application descriptions to help other AI application developers better understand and use your applications. Click **Add AI Application Description** and set the **Document name** and **URL**. You can add up to three AI application descriptions.
Deployment Type	Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select **Real-time services** here, you can only deploy the AI application as a real-time service after it is created.

c. Check the information and click **Next**. The AI application is created.

In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is successfully created. On this page, you can perform such operations as creating new versions and quickly deploying services.

Follow-Up Procedure

Deploying an AI Application as a Service: In the AI application list, click the option button on the left of the AI application name to display the version list at the bottom of the list page. Locate the row that contains the target version, click **Deploy** in the **Operation** column to deploy the AI application as a service type selected during AI application creation.

2.2.3 Importing a Meta Model from OBS

In scenarios where frequently-used frameworks are used for model development and training, you can import the model to ModelArts and use it to create an AI application for unified management.

Constraints

- The imported model for creating an AI application, inference code, and configuration file must comply with the requirements of ModelArts. For details, see **Introduction to Model Package Specifications**, **Specifications for Editing a Model Configuration File** , and **Specifications for Writing Model Inference Code** .

- If the meta model is from a container image, ensure the size of the meta model complies with **Restrictions on the Size of an Image for Importing an AI Application**.

Prerequisites

- The model has been developed and trained, and the type and version of the AI engine used by the model are supported by ModelArts. For details, see **Supported AI Engines for ModelArts Inference**.

- The trained model package, inference code, and configuration file have been uploaded to OBS.

- The OBS directory you use and ModelArts are in the same region.

Creating an AI Application

1. Log in to the ModelArts management console, and choose **AI Application Management** > **AI Applications** in the left navigation pane. The **AI Applications** page is displayed.

2. Click **Create** in the upper left corner.

3. On the displayed page, set the parameters.

   a. Set basic information about the AI application. For details about the parameters, see **Table 2-7**.

   Table 2-7 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Version	Version of the AI application to be created. For the first import, the default value is **0.0.1**. **NOTE** After an AI application is created, you can **create new versions** using different meta models for optimization.
Description	Brief description of an AI application

   b. Select the meta model source and set related parameters. Set **Meta Model Source** to **OBS**. For details about the parameters, see **Table 2-8**.

   For the meta model imported from OBS, edit the inference code and configuration files by following **model package specifications** and place the inference code and configuration files in the **model** folder storing the

meta model. If the selected directory does not comply with the model package specifications, the AI application cannot be created.

Table 2-8 Parameters of the meta model source

Parameter	Description
Meta Model	OBS path for storing the meta model.
	The OBS path cannot contain spaces. Otherwise, the AI application fails to be created.
AI Engine	The AI engine automatically associates with the meta model storage path you select.
	If **AI Engine** is set to **Custom**, you must specify the protocol and port number in **Container API** for starting the model. The request protocol is **HTTPS**, and the port number is **8080**.

Parameter	Description
Health Check	Health check on a model. After you select an AI engine that supports health check and runtime environment, this parameter is displayed. When **AI Engine** is set to **Custom**, you must configure health check in the image. Otherwise, the service deployment will fail. ● **Check Mode**: Select **HTTP request** or **Command**. When a custom engine is used, you can select **HTTP request** or **Command**. When a non-custom engine is used, you can select only **HTTP request**. ● **Health Check URL**: This parameter is displayed when **Check Mode** is set to **HTTP request**. Enter the health check URL. The default value is **/health**. ● **Health Check Command**: This parameter is displayed when **Check Mode** is set to **Command**. Enter the health check command. ● **Health Check Period**: Enter an integer ranging from 1 to 2147483647. The unit is second. ● **Delay(seconds)**: specifies the delay for performing the health check after the instance is started. Enter an integer ranging from 0 to 2147483647. ● **Maximum Failures**: Enter an integer ranging from 1 to 2147483647. During service startup, if the number of consecutive health check failures reaches the specified value, the service will be abnormal. During service running, if the number of consecutive health check failures reaches the specified value, the service will enter the alarm status. **NOTE** To use a custom engine to create an AI application, ensure that the custom engine complies with the specifications for custom engines. For details, see **Creating an AI Application Using a Custom Engine**. If health check is configured for an AI application, the deployed services using this AI application will stop 3 minutes after receiving the stop instruction.
Dynamic Loading	Quick deployment and model update. If it is selected, model files and runtime dependencies are only pulled during an actual deployment. Enable this function if a single model file is larger than 5 GB.
Runtime Dependency	List the dependencies of the selected model in the environment. For example, if **tensorflow** is used and the installation method is **pip**, the version must be 1.8.0 or later.

Parameter	Description
AI Application Description	Provide AI application descriptions to help other AI application developers better understand and use your applications. Click **Add AI Application Description** and set the **Document name** and **URL**. You can add up to three AI application descriptions.
Configuration File	By default, the system associates the configuration file stored in OBS. After enabling this function, you can view and edit the model configuration file. **NOTE** This function is to be taken offline. After that, you can modify the model configuration by setting **AI Engine**, **Runtime Dependency**, and **Apis**.
Deployment Type	Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select **Real-time services** here, you can only deploy the AI application as a real-time service after it is created.
API Configuration	After enabling this function, you can edit RESTful APIs to define the input and output formats of an AI application. The model APIs must comply with ModelArts specifications. For details, see **Specifications for Editing a Model Configuration File**. For details about the code example, see **Code Example of apis Parameters**.

    c.   Check the information and click **Create now**. The AI application is created.

        In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is successfully created. On this page, you can perform such operations as creating new versions and quickly deploying services.

Follow-Up Procedure

Deploying an AI Application as a Service: In the AI application list, click the option button on the left of the AI application name to display the version list at the bottom of the list page. Locate the row that contains the target version, click **Deploy** in the **Operation** column to deploy the AI application as a service type selected during AI application creation.

2.2.4 Importing a Meta Model from a Container Image

For AI engines that are not supported by ModelArts, you can import the models you compile to ModelArts from custom images.

Constraints

- For details about the specifications and description of custom images, see **Custom Image Specifications for Creating AI Applications**.

- The configuration file must be provided for a model that you have developed and trained. The file must comply with ModelArts specifications. For details, see **Specifications for Editing a Model Configuration File** . After the writing is completed, upload the file to the specified OBS directory.
- If the meta model is from a container image, ensure the size of the meta model complies with **Restrictions on the Size of an Image for Importing an AI Application**.

Prerequisites

The OBS directory you use and ModelArts are in the same region.

Creating an AI Application

1. Log in to the ModelArts management console, and choose **AI Application Management** > **AI Applications** in the left navigation pane. The **AI Applications** page is displayed.
2. Click **Create** in the upper left corner.
3. On the displayed page, set the parameters.

   a. Set basic information about the AI application. For details about the parameters, see **Table 2-9**.

   Table 2-9 Parameters of basic AI application information

Parameter	Description
Name	Application name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.
Version	Version of the AI application to be created. For the first import, the default value is **0.0.1**. **NOTE** After an AI application is created, you can **create new versions** using different meta models for optimization.
Description	Brief description of an AI application

   b. Select the meta model source and set related parameters. Set **Meta Model Source** to **Container image**. For details about the parameters, see **Table 2-10**.

Figure 2-2 Setting a container image as the meta model source
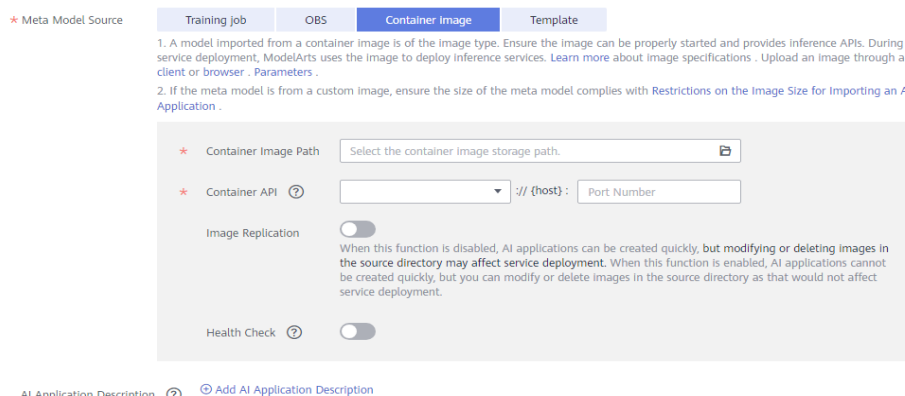
Table 2-10 Parameters of the meta model source

Parameter	Description
Container Image Path	Click ☐ to import the model image from the container image. The model is of the Image type, and you do not need to use **swr_location** in the configuration file to specify the image location. For details about operation guidance and requirements for creating a custom image, see **Custom Image Specifications for Creating AI Applications**. **NOTE** The model image you select will be shared with the system administrator, so ensure you have the permission to share the image (images shared with other accounts are not supported). When you deploy a service, ModelArts deploys the image as an inference service. Ensure that your image can be properly started and provide an inference API.
Container API	Protocol and port number for starting an AI application **NOTE** The default request protocol and port number provided by ModelArts are HTTP and 8080, respectively. Set them based on the actual custom image.

Parameter	Description
Image Replication	Indicates whether to copy the model image in the container image to ModelArts. ● When this function is disabled, the model image is not copied, AI applications can be created quickly, but modifying or deleting images in the source directory of SWR may affect service deployment. ● When this function is enabled, the model image is copied, AI applications cannot be created quickly, but you can modify or delete images in the source directory of SWR as that would not affect service deployment. **NOTE** You must enable this function if you want to use images shared by others. Otherwise, AI applications will fail to be created.
Health Check	Health check on an AI application. This parameter is configurable only when the health check API is configured in the custom image. Otherwise, the AI application deployment will fail. ● **Check Mode**: Select **HTTP request** or **Command**. ● **Health Check URL**: This parameter is displayed when **Check Mode** is set to **HTTP request**. Enter the health check URL. The default value is **/health**. ● **Health Check Command**: This parameter is displayed when **Check Mode** is set to **Command**. Enter the health check command. ● **Health Check Period**: Enter an integer ranging from 1 to 2147483647. The unit is second. ● **Delay(seconds)**: specifies the delay for performing the health check after the instance is started. Enter an integer ranging from 0 to 2147483647. ● **Maximum Failures**: Enter an integer ranging from 1 to 2147483647. During service startup, if the number of consecutive health check failures reaches the specified value, the service will be abnormal. During service running, if the number of consecutive health check failures reaches the specified value, the service will enter the alarm status. **NOTE** If health check is configured for an AI application, the deployed services using this AI application will stop 3 minutes after receiving the stop instruction.

Parameter	Description
AI Application Description	Provide AI application descriptions to help other AI application developers better understand and use your applications. Click **Add AI Application Description** and set the **Document name** and **URL**. You can add up to three AI application descriptions.
Deployment Type	Select the service types that the application can be deployed. When deploying a service, only the service types selected here are available. For example, if you only select **Real-time services** here, you can only deploy the AI application as a real-time service after it is created.
Start command	Customizable start command of a model
Apis	When you enable this function, you can edit RESTful APIs to define the AI application input and output formats. The model APIs must comply with ModelArts specifications. For details, see **Specifications for Editing a Model Configuration File**. For details about the code example, see **Code Example of apis Parameters**.

c. Check the information and click **Next**. The AI application is created.

In the AI application list, you can view the created AI application and its version. When the status changes to **Normal**, the AI application is successfully created. On this page, you can perform such operations as creating new versions and quickly deploying services.

Follow-Up Procedure

Deploying an AI Application as a Service: In the AI application list, click the option button on the left of the AI application name to display the version list at the bottom of the list page. Locate the row that contains the target version, click **Deploy** in the **Operation** column to deploy the AI application as a service type selected during AI application creation.

2.3 Viewing the AI Application List

You can view all created AI applications on the AI application list page. The AI application list page displays the following information.

Table 2-11 AI application list

Parameter	Description
AI Application Name	Name of an AI application.
Latest Version	Latest version of an AI application.

Parameter	Description
Status	Status of an AI application.
Deployment Type	Types of the services that an AI application can be deployed as.
Versions	Number of AI application versions.
Request Mode	Request mode of real-time services. ● Synchronous request: one-off inference with results returned synchronously (within 60s). This mode is suitable for images and small videos. ● Asynchronous request: one-off inference with results returned asynchronously (over 60s). This mode is suitable for real-time video inference and large videos.
Created	Time when an AI application is created.
Description	Description of an AI application.

Operation	● **Create Version**: Create an AI application version. The settings of the last version are used by default, except for the version. You can change the parameter settings.
	● **Delete**: Delete the AI application.
	   NOTE
	   If an AI application version has been deployed as a service, you must delete the associated service before deleting the AI application version. A deleted AI application cannot be recovered.

Click the check box next to the AI application name to display the hidden view at the bottom of the list, where you can see the version list. (If the view is not

displayed, click ⌃ in the bottom right corner.)

Figure 2-3 Version list
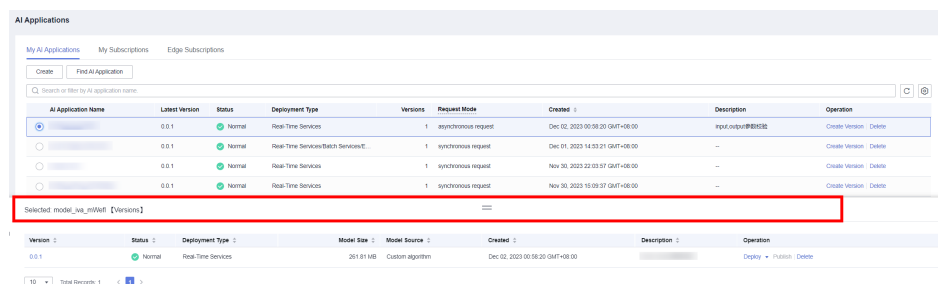
The version list displays the following information.

Table 2-12 Version list

Parameter	Description
Version	Current version of an AI application.
Status	Status of an AI application.
Deployment Type	Types of the services that an AI application can be deployed as.
Model Size	Size of an AI application.
Model Source	Model source of an AI application.
Created	Time when an AI application is created.
Description	Description of an AI application.
Operation	● **Deploy**: Deploy an AI application as real-time services, batch services, or edge services. ● **Delete**: Delete a version of an AI application.

2.4 Viewing Details About an AI Application

After an AI application is created, you can view its information on the details page.

1. Log in to the ModelArts console, and choose **AI Applications** from the navigation pane. The **My AI Applications** tab is displayed by default.
2. Click the name of the target AI application. The application details page is displayed.

   On the application details page, you can view the basic information and model precision of the AI application, and switch tab pages to view more information.

Table 2-13 Basic information about an AI application

Parameter	Description
Name	Name of an AI application
Status	Status of an AI application
Version	Current version of an AI application
ID	ID of an AI application
Description	Click the edit button to add the description of an AI application.
Deployment Type	Types of the services that an AI application can be deployed

Parameter	Description
Meta Model Source	Source of the meta model, which can be training jobs, OBS, or container images.
Training Name	Associated training job if the meta model comes from a training job. Click the training job name to go to its details page.
Training Version	Training job version if the meta model comes from an old-version training job.
Storage path of the meta model	Path to the meta model if the meta model comes from OBS.
Container Image Storage Path	Path to the container image if the meta model comes from a container image.
AI Engine	AI engine if the meta model comes from a training job or OBS.
Engine Package Address	Engine package address if the meta model comes from OBS and **AI Engine** is **Custom**.
Runtime Environment	Runtime environment on which the meta model depends if the meta model comes from a training job or OBS and a preset AI engine is used.
Container API	Protocol and port number for starting the AI application if the meta model comes from OBS (**AI Engine** is **Custom**) or a container image.
Inference Code	Path to the inference code if the meta model comes from an olde-version training job.
Image Replication	Image replication status if the meta model comes from OBS or a container image.
Dynamic loading	Dynamic loading status if the meta model comes from a training job or OBS.
Size	Size of an AI application
Health Check	Health check status if the meta model comes from OBS or a container image. If health check is enabled, the following parameters are displayed: **Check Mode**, **Health Check URL**, **Health Check Period**, **Delay**, and **Maximum Failures**.
AI Application Description	Description document added during the creation of an AI application.
Instruction Set Architecture	System architecture.
Inference Accelerator	Type of inference accelerator cards.

Table 2-14 Details page of an AI application

Parameter	Description
Model Precision	Model recall, precision, accuracy, and F1 score of an AI application
Parameter Configuration	API configuration, input parameters, and output parameters of an AI application
Runtime Dependency	Model dependency on the environment. If creating a job failed, edit the runtime dependency. After the modification is saved, the system will automatically use the original image to create the job again.
Events	The progress of key operations during AI application creation

Events are stored for three months and will be automatically cleared then.

For details about how to view events of an AI application, see **Viewing Events of an AI Application**. |
| Constraint | Displays the constraints of service deployment, such as the request mode, boot command, and model encryption, based on the settings during AI application creation. For AI applications in asynchronous request mode, parameters including the input mode, output mode, service startup parameters, and job configuration parameters can be displayed. |
| Associated Services | The list of services that an AI application was deployed. Click a service name to go to the service details page. |

2.5 Managing AI Application Versions

To facilitate source tracing and repeated AI application tuning, ModelArts provides the AI application version management function. You can manage models based on versions.

Prerequisites

An AI application has been created in ModelArts.

Creating a New Version

On the **AI Application Management** > **AI Applications** page, click **Create Version** in the **Operation** column of the target AI application. On the **Create Version** page, set the parameters. For details, see **Creating an AI Application**. Click **Create now**.

Deleting a Version

On the **AI Application Management > AI Applications** page, click the option button on the left of the AI application name to display the application version list. In the application version list, click **Delete** in the **Operation** column to delete the corresponding version.

📖 **NOTE**

If a service has been deployed for the AI application version, you need to delete the associated service before deleting the AI application version. A deleted version cannot be recovered. Exercise caution when performing this operation.

Deleting an AI Application

In the navigation pane, choose **AI Application Management** > **AI Applications**. On the **AI Applications** page, click **Delete** in the **Operation** column to delete the target AI application.

📖 **NOTE**

If a service has been deployed for the AI application version, you need to delete the associated service before deleting the AI application version. A deleted AI application cannot be recovered. Exercise caution when performing this operation.

2.6 Viewing Events of an AI Application

During the creation of an AI application, every key event is automatically recorded. You can view the events on the details page of the AI application at any time.

This helps you better understand the process of creating an AI application and locate faults more accurately when a task exception occurs. The following table lists the available events.

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	The model starts to import.	-
Abnormal	Failed to create the image.	Locate and rectify the fault based on the error information. **FAQ**
Abnormal	The custom image does not support specified dependencies.	The runtime dependencies cannot be configured when a custom image is imported. Install the pip dependency package in the Dockerfile that is used to create the image. **FAQ**

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	Only custom images support **swr_location**.	Delete the **swr_location** field from the model configuration file **config.json** and try again.
Abnormal	The health check API of a custom image must be *xxx*.	Modify the health check API of the custom image and try again.
Normal	The image creation task is in the *xxx* state.	-
Abnormal	Label *xxx* does not exist in image *xxx*.	Contact technical support.
Abnormal	Invalid parameter value *xxx* exists in the model configuration file.	Delete invalid parameters from the model configuration file and try again.
Abnormal	Failed to obtain the labels of image *xxx*.	Contact technical support.
Abnormal	Failed to import data because *xxx* is larger than *xxx* GB.	The size of the model or image exceeds the upper limit. Downsize the model or image and import it again. **FAQ**
Abnormal	User *xxx* does not have OBS permission obs:object:PutObjectAcl.	The IAM user does not have the obs:object:PutObject Acl permission on OBS. Add the agency permission for the IAM user. **FAQ**
Abnormal	Creating the image timed out. The timeout duration is *xxx* minutes.	There is a timeout limit for image building using ImagePacker. Simplify the code to improve efficiency. **FAQ**
Normal	Model description updated.	-

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	Model runtime dependencies not updated.	-
Normal	Model runtime dependencies updated. Recreating the image.	-
Abnormal	SWR traffic control triggered. Try again later.	SWR traffic control triggered. Try again later.
Normal	The system is being upgraded. Try again later.	-
Abnormal	Failed to obtain the source image. An error occurred in authentication. The token has expired.	Contact technical support.
Abnormal	Failed to obtain the source image. Check whether the image exists.	Contact technical support.
Normal	Source image size calculated.	-
Normal	Source image shared.	-
Abnormal	Failed to create the image due to traffic control. Try again later.	Traffic control triggered. Try again later.
Abnormal	Failed to send the image creation request.	Contact technical support.
Abnormal	Failed to share the source image. Check whether the image exists or whether you have the permission to share the image.	Check whether the image exists or whether you have the permission to share the image.
Normal	The model imported.	-
Normal	Model file imported.	-
Normal	Model size calculated.	-
Abnormal	Failed to import the model.	For details about how to locate and rectify the fault, see **FAQ**.
Abnormal	Failed to copy the model file. Check whether you have the OBS permission.	Check whether you have the OBS permission. **FAQ**
Abnormal	Failed to schedule the image creation task.	Contact technical support.

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	Failed to start the image creation task.	Contact technical support.
Abnormal	The Roman image has been created but cannot be shared with resource tenants.	Contact technical support.
Normal	Image created.	-
Normal	The image creation task started.	-
Normal	The environment image creation task started.	-
Normal	The request for creating an environment image received.	-
Normal	The request for creating an image received.	-
Normal	An existing environment image is used.	-
Abnormal	Failed to create the image. For details, see image creation logs.	View the build logs to locate and rectify the fault. **FAQ**
Abnormal	Failed to create the image due to an internal system error. Contact technical support.	Contact technical support.
Abnormal	Failed to import model file *xxx* because it is larger than 5 GB.	The size of the model file *xxx* is greater than 5 GB. Downsize the model file and try again, or use dynamic loading to import the model file. **FAQ**
Abnormal	Failed to create the OBS bucket due to an internal system error. Contact technical support.	Contact technical support.
Abnormal	Failed to calculate the model size. Subpath *xxx* does not exist in path *xxx*.	Correct the subpath and try again, or contact technical support.

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	Failed to calculate the model size. The model of the *xxx* type does not exist in path *xxx*.	Check the storage location of the model of the *xxx* type, correct the path, and try again, or contact technical support.
Warning	Failed to calculate the model size. More than one *xxx* model file is stored in path *xxx*.	-

During AI application creation, key events can both be manually and automatically refreshed.

Viewing Events

1. In the navigation pane of the ModelArts management console, choose **AI Application Management** > **AI Applications**. In the AI application list, click the name of the target AI application to go to its details page.
2. View the events on the **Events** tab page.

3 Deploying an AI Application as a Service

Deploying AI Applications as Real-Time Services

Deploying AI Applications as Batch Services

Upgrading a Service

Starting, Stopping, Deleting, or Restarting a Service

Viewing Service Events

3.1 Deploying AI Applications as Real-Time Services

3.1.1 Deploying as a Real-Time Service

After an AI application is prepared, you can deploy it as a real-time service and call the service for prediction.

Constraints

A maximum of 20 real-time services can be deployed by a user.

Prerequisites

- Data has been prepared. Specifically, you have created an AI application in the **Normal** state in ModelArts.
- The account is not in arrears to ensure available resources for service running.

Procedure

1. Log in to the ModelArts management console. In the left navigation pane, choose **Service Deployment** > **Real-Time Services**. The real-time service list is displayed by default.
2. In the real-time service list, click **Deploy** in the upper left corner. The **Deploy** page is displayed.
3. Set parameters for a real-time service.

a. Set basic information about model deployment. For details about the parameters, see **Table 3-1**.

Table 3-1 Basic parameters

Parameter	Description
Name	Name of the real-time service. Set this parameter as prompted.
Auto Stop	After this parameter is enabled and the auto stop time is set, a service automatically stops at the specified time. If this parameter is disabled, a real-time service keeps running and billing. The function can help you avoid unnecessary billing. The auto stop function is enabled by default, and the default value is **1 hour later**. The options are **1 hour later**, **2 hours later**, **4 hours later**, **6 hours later**, and **Custom**. If you select **Custom**, you can enter any integer from 1 to 24 hours in the text box on the right.
Description	Brief description of the real-time service.

b. Enter key information including the resource pool and AI application configurations. For details, see **Table 3-2**.

Table 3-2 Parameters

Parameter	Sub-Parameter	Description
Resource Pool	Public Resource Pool	CPU/GPU computing resources are available for you to select. Pricing standards for resource pools with different flavors are different. For details, see **Product Pricing Details**. The public resource pool only supports the pay-per-use billing mode.

Param eter	Sub-Parame ter	Description
	Dedicat ed Resourc e Pool	Select a specification from the dedicated resource pool specifications. The physical pools with logical subpools created are not supported temporarily. **NOTE** ● The data of old-version dedicated resource pools will be gradually migrated to the new-version dedicated resource pools. ● For new users and the existing users who have migrated data from old-version dedicated resource pools to new ones, there is only one entry to new-version dedicated resource pools on the ModelArts management console. ● For the existing users who have not migrated data from old-version dedicated resource pools to new ones, there are two entries to dedicated resource pools on the ModelArts management console, where the entry marked with **New** is to the new version. For details about the new version of dedicated resource pools, see Comprehensive Upgrades to ModelArts Resource Pool Management Functions.
AI Applic ation and Config uration	AI Applicat ion Source	Select **My AI Applications** or **My Subscriptions** based on your requirements.
	AI Applicat ion and Version	Select the AI application and version that are in the **Normal** state.
	Traffic Ratio (%)	Set the traffic proportion of the current instance node. Service calling requests are allocated to the current version based on this proportion. If you deploy only one version of an AI application, set this parameter to **100%**. If you select multiple versions for gray release, ensure that the sum of the traffic ratios of these versions is **100%**.

Param eter	Sub- Parame ter	Description
	Specific ations	Select available specifications based on the list displayed on the console. The specifications in gray cannot be used in the current environment. If specifications in the public resource pools are unavailable, no public resource pool is available in the current environment. In this case, use a dedicated resource pool or contact the administrator to create a public resource pool. **NOTE** When the selected flavor is used to deploy the service, necessary system consumption is generated. Therefore, the resources actually occupied by the service are slightly greater than the selected flavor.
	Comput e Nodes	Set the number of instances for the current AI application version. If you set the number of nodes to **1**, the standalone computing mode is used. If you set the number of nodes to a value greater than 1, the distributed computing mode is used. Select a computing mode based on the actual requirements.
	Environ ment Variable	Set environment variables and inject them to the pod. To ensure data security, do not enter sensitive information such as plaintext passwords in environment variables.
	Timeout	Timeout of a single model, including both the deployment and startup time. The default value is 20 minutes. The value must range from 3 to 120.
	Add AI Applicat ion Version and Configu ration	If the selected AI application has multiple versions, you can add multiple versions and configure a traffic ratio. You can use gray launch to smoothly upgrade the AI application version. **NOTE** Free compute specifications do not support the gray launch of multiple versions.

Param eter	Sub- Parame ter	Description
	Mount Storage	This parameter is displayed when the resource pool is a dedicated resource pool. This function will mount a storage volume to compute nodes (compute instances) as a local directory when the service is running. It is recommended when the model or input data is large. Only OBS parallel file systems are supported. ● **Source Path**: Select the storage path of the parallel file. A cross-region OBS parallel file system cannot be selected. ● **Mount Path**: Enter the mount path of the container, for example, **/obs-mount/**. – Select a new directory. If you select an existing directory, existing files will be overwritten. OBS mounting allows you to add, view, and modify files in the mount directory but does not allow you to delete files in the mount directory. To delete files, manually delete them in the OBS parallel file system. – It is a good practice to mount the container to an empty directory. If the directory is not empty, ensure that there are no files affecting container startup in the directory. Otherwise, such files will be replaced, resulting in failures to start the container and create the workload. – The mount path must start with a slash (/) and can contain a maximum of 1,024 characters, including letters, digits, and the following special characters: \ _ -. NOTE Storage mounting can be used only by services deployed in a dedicated resource pool.
Traffic Limit	N/A	Maximum number of times a service can be accessed within a second. You can set this parameter as needed.

Param eter	Sub- Parame ter	Description
WebSo cket	N/A	Whether to deploy a real-time service as a WebSocket service. For details about WebSocket real-time services, see **Full-Process Development of WebSocket Real-Time Services**. **NOTE** ● This function is supported only if the AI application is WebSocket-compliant and comes from a container image. ● After this function is enabled, **Traffic Limit** and **Data Collection** cannot be set. ● This parameter cannot be changed after the service is deployed.
Runtim e Log Output	N/A	This function is disabled by default. The runtime logs of real-time services are stored only in the ModelArts log system. You can query the runtime logs on the **Logs** tab of the service details page. If this function is enabled, the runtime logs of real-time services will be exported and stored in Log Tank Service (LTS). LTS automatically creates log groups and log streams and caches run logs generated within seven days by default. For details about the LTS log management function, see **Log Tank Service**. **NOTE** ● This cannot be disabled once it is enabled. ● You will be billed for the log query and log storage functions provided by LTS. For details, see section **LTS Pricing Details**. ● Do not print unnecessary audio log files. Otherwise, system logs may fail to be displayed, and the error message "Failed to load audio" may be displayed.
Applic ation Authen ticatio n	Applicat ion	Disabled by default. To enable this function, see **Access Authenticated Using an Application** for details and set the parameters as required.

Figure 3-1 Setting AI application information
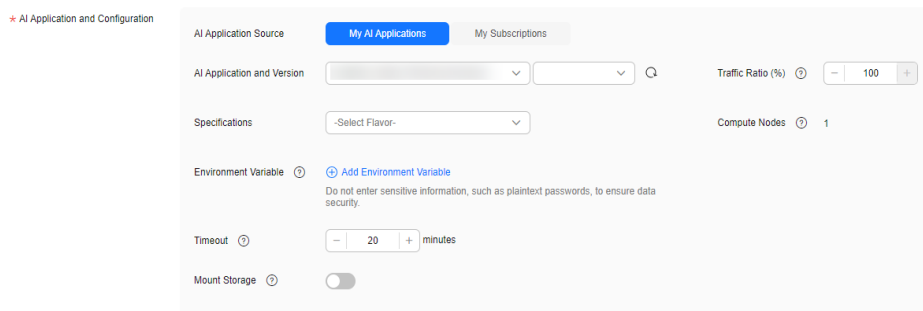
c.　(Optional) Configure advanced settings.

Table 3-3 Advanced settings

Parameter	Description
Tags	ModelArts can work with Tag Management Service (TMS). When creating resource-consuming tasks in ModelArts, for example, training jobs, configure tags for these tasks so that ModelArts can use tags to manage resources by group.
	For details about how to use tags, see **How Does ModelArts Use Tags to Manage Resources by Group?**
	NOTE You can select a predefined TMS tag from the tag drop-down list or customize a tag. Predefined tags are available to all service resources that support tags. Customized tags are available only to the service resources of the user who has created the tags.

4.　After confirming the entered information, complete service deployment as prompted. Generally, service deployment jobs run for a period of time, which may be several minutes or tens of minutes depending on the amount of your selected data and resources.

📖 **NOTE**

After a real-time service is deployed, it is started immediately.

You can go to the real-time service list to check whether the deployment of the real-time service is complete. In the real-time service list, after the status of the newly deployed service changes from **Deploying** to **Running**, the service is deployed successfully.

3.1.2 Viewing Service Details

After an AI application is deployed as a real-time service, you can access the service page to view its details.

1.　Log in to the ModelArts management console and choose **Service Deployment** > **Real-Time Services**.

2. On the **Real-Time Services** page, click the name of the target service. The service details page is displayed.

You can view the service name, status, and other information. For details, see **Table 3-4**.

Table 3-4 real-time service parameters

Parameter	Description
Name	Name of the real-time service.
Status	Status of the real-time service.
Source	AI application source of the real-time service.
Service ID	Real-time service ID
Description	Service description, which can be edited after you click the edit button on the right side.
Resource Pool	Resource pool specifications used by the service. If the public resource pool is used for deployment, this parameter is not displayed.
Custom Settings	Customized configurations based on real-time service versions. This allows version-based traffic distribution policies and configurations. Enable this option and click **View Settings** to customize the settings. For details, see **Modifying Customized Settings**.
Traffic Limit	Maximum number of times a service can be accessed within a second.
Runtime Log Output	This function is disabled by default. The runtime logs of real-time services are stored only in the ModelArts log system. If this function is enabled, the runtime logs of real-time services will be exported and stored in Log Tank Service (LTS). LTS automatically creates log groups and log streams and caches run logs generated within seven days by default. For details about the LTS log management function, see **Log Tank Service**. NOTE ● This cannot be disabled once it is enabled. ● You will be billed for the log query and log storage functions provided by LTS. For details, see section **LTS Pricing Details**. ● Do not print unnecessary audio log files. Otherwise, system logs may fail to be displayed, and the error message "Failed to load audio" may be displayed.
WebSocket	Whether to upgrade to the WebSocket service.

3. Switch between tabs on the details page of a real-time service to view more details. For details, see **Table 3-5**.

Table 3-5 Details of a real-time service

Parameter	Description
Usage Guides	This page displays the API URL, AI application information, input parameters, and output parameters. You can click ▢ to copy the API URL to call the service. If application authentication is supported, you can view the API URL and authorization management details, including the application name, AppKey, and AppSecret, in the **Usage Guides**. You can also add or cancel authorization for an application.
Prediction	You can perform real-time prediction on this page. For details, see **Testing the Deployed Service**.
Configuration Updates	This page displays **Current Configurations** and **Update History**. ● **Current Configurations**: AI application name, version, status, compute node specifications, traffic ratio, number of compute nodes, deployment timeout interval, environment variables, storage mounting, and resource pool information (for services deployed in a dedicated resource pool) ● **Update History**: historical AI application information.
Monitoring	This page displays resource usage and AI application calls. ● **Resource Usage**: includes the used and available CPU, memory, GPU, and NPU resources. ● **AI Application Calls**: indicates the number of AI application calls. The statistics collection starts after the AI application status changes to **Ready**. (This parameter is not displayed for WebSocket services.)
Event	This page displays key operations during service use, such as the service deployment progress, detailed causes of deployment exceptions, and time points when a service is started, stopped, or modified. Events are saved for one month and will be automatically cleared then. For details about how to view events of a service, see **Viewing Service Events**.

Parameter	Description
Logs	This page displays the log information about each AI application in the service. You can view logs generated in the latest 5 minutes, latest 30 minutes, latest 1 hour, and user-defined time segment.
	You can select the start time and end time when defining the time segment.
	If this function is enabled, the logs stored in LTS will be displayed. You can click **View Complete Logs on LTS** to view all logs.
	Meet the following rules to search logs:
	• Do not enter strings that contain any following delimiters: **,'";=()[]{}@&<>/:\n\t\r**.
	• Enter keywords for exact search. A keyword is a word between two adjacent delimiters.
	• Enter keywords for fuzzy search. For example, you can enter **error**, **er?or**, **rro***, or **er*r**.
	• Enter phrases for exact search. For example, **Start to refresh**.
	• Before enabling this function, you can combine keywords with AND (**&&**) or OR (**||**). For example, **query logs&&erro*** or **query logs||erro***. After enabling this function, you can combine keywords with **AND** or **OR**. For example, **query logs AND erro*** or **query logs OR erro***.
Tags	Tags that have been added to the service. Tags can be added, modified, and deleted.
	For details about how to use tags, see **How Does ModelArts Use Tags to Manage Resources by Group?**

Modifying Customized Settings

A customized configuration rule consists of the configuration condition (**Setting**), access version (**Version**), and customized running parameters (including **Setting Name** and **Setting Value**).

You can configure different settings with customized running parameters for different versions of a real-time service.

The priorities of customized configuration rules are in descending order. You can change the priorities by dragging the sequence of customized configuration rules.

After a rule is matched, the system will no longer match subsequent rules. A maximum of 10 configuration rules can be configured.

Table 3-6 Parameters for **Custom Settings**

Parameter	Mandatory	Description
Setting	Yes	Expression of the Spring Expression Language (SPEL) rule. Only the equal, matches, and hashCode expressions of the character type are supported.
Version	Yes	Access version for a customized service configuration rule. When a rule is matched, the real-time service of the version is requested.
Setting Name	No	Key of a customized running parameter, consisting of a maximum of 128 characters.

Configure this parameter if the HTTP message header is used to carry customized running parameters to a real-time service. |
| Setting Value | No | Value of a customized running parameter, consisting of a maximum of 256 characters.

Configure this parameter if the HTTP message header is used to carry customized running parameters to a real-time service. |

Customized settings can be used in the following scenarios:

- If multiple versions of a real-time service are deployed for gray release, customized settings can be used to distribute traffic by user.

Table 3-7 Built-in variables

Built-in Variable	Description
DOMAIN_NAME	Account name used to call an inference request
DOMAIN_ID	Account ID used to call an inference request
PROJECT_NAME	Project name that is used to call an inference request
PROJECT_ID	Project ID that invokes the inference request
USER_NAME	Username that is used to call an inference request
USER_ID	User ID that is used to call an inference request

Pound key (#) indicates that a variable is referenced. The matched character string must be enclosed in single quotation marks.

```
#{Built-in variable} == 'Character string'
#{Built-in variable} matches 'Regular expression'
```

– Example 1:

If the account name in the inference request is **User A**, the specified version is matched.

```
#DOMAIN_NAME == 'User A'
```

– Example 2:

If the account name in the inference request starts with **op**, the specified version is matched.

```
#DOMAIN_NAME matches 'op.*'
```

Table 3-8 Common regular expressions

Character	Description
.	Match any single character except **\n**. To match any character including **\n**, use **(.|\n)**.
*	Match the subexpression that it follows for zero or multiple times. For example, **zo*** can match **z** and **zoo**.
+	Match the subexpression that it follows for once or multiple times. For example, **zo+** can match **zo** and **zoo**, but cannot match **z**.
?	Match the subexpression that it follows for zero or one time. For example, **do(es)?** can match **does** or **do** in **does**.
^	Match the start of the input string.
$	Match the end of the input string.
{n}	*n* is a non-negative integer, which matches exactly *n* number of occurrences of an expression. For example, **o{2}** cannot match **o** in **Bob**, but can match two **o**s in **food**.
x|y	Match x or y. For example, **z|food** can match **z** or **food**, and **(z|f)ood** can match **zood** or **food**.
[xyz]	Character set, where any single character in it can be matched. For example, **[abc]** can match **a** in **plain**.

Figure 3-2 Traffic distribution by user
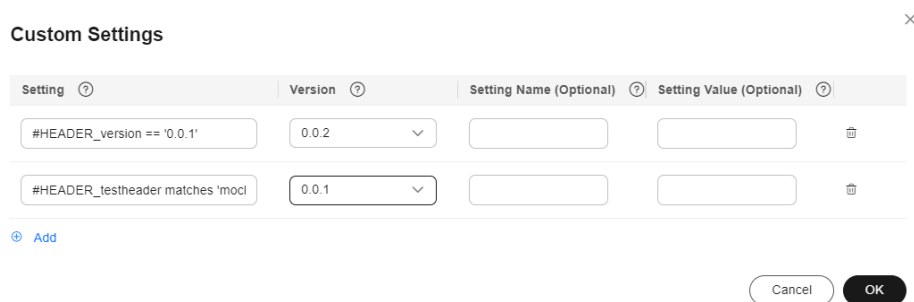
- If multiple versions of a real-time service are deployed for gated launch, customized settings can be used to access different versions through the header.

  Start with **#HEADER_** to indicate that the header is referenced as a condition.

  ```
  #HEADER_{key} == '{value}'
  #HEADER_{key} matches '{value}'
  ```

  - Example 1:

    If the header of an inference HTTP request contains a version and the value is **0.0.1**, the condition is met. Otherwise, the condition is not met.

    ```
    #HEADER_version == '0.0.1'
    ```

  - Example 2:

    If the header of an inference HTTP request contains **testheader** and the value starts with **mock**, the rule is matched.

    ```
    #HEADER_testheader matches 'mock.*'
    ```

  - Example 3:

    If the header of an inference HTTP request contains **uid** and the hash code value meets the conditions described in the following algorithm, the rule is matched.

    ```
    #HEADER_uid.hashCode() % 100 < 10
    ```

  Figure 3-3 Using the header to access different versions
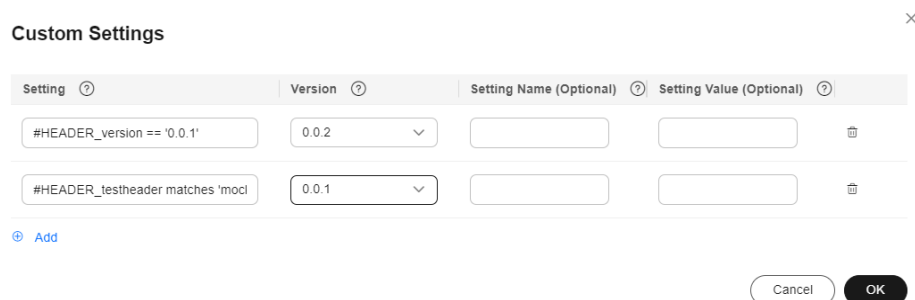
- If a real-time service version supports different runtime configurations, you can use **Setting Name** and **Setting Value** to specify customized runtime parameters so that different users can use different running configurations.

  Example:

  When user A accesses the AI application, the user uses configuration A. When user B accesses the AI application, the user uses configuration B. When matching a running configuration, ModelArts adds a header to the request and also the customized running parameters specified by **Setting Name** and **Setting Value**.

Figure 3-4 Customized running parameters added for a customized configuration rule
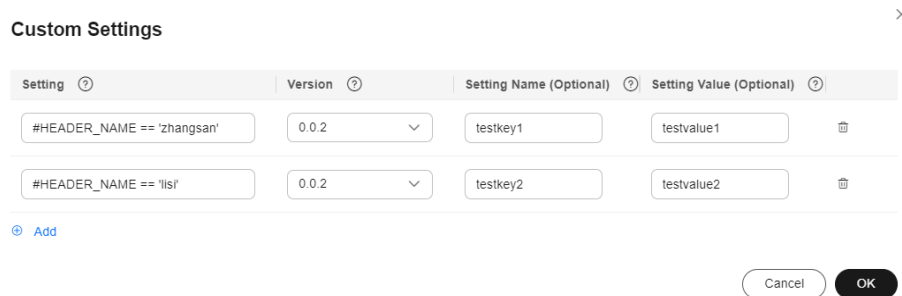
3.1.3 Testing the Deployed Service

After an AI application is deployed as a real-time service, you can debug code or add files for testing on the **Prediction** tab page. Based on the input request (JSON text or file) defined by the AI application, the service can be tested in either of the following ways:

- **JSON Text Prediction**: If the input type of the AI application of the deployed service is JSON text, that is, the input does not contain files, you can enter the JSON code on the **Prediction** tab page for service testing.

- **File Prediction**: If the input type of the AI application of the deployed service is file, including images, audios, and videos, you can add images on the **Prediction** tab page for service testing.

📖 NOTE

- If the input type is image, the size of a single image must be less than 8 MB.
- The maximum size of the request body for JSON text prediction is 8 MB.
- Due to the limitation of API Gateway, the duration of a single prediction cannot exceed 40s.
- The following image types are supported: png, psd, jpg, jpeg, bmp, gif, webp, psd, svg, and tiff.
- If Ascend flavors are used during service deployment, transparent .png images cannot be predicted because Ascend supports only RGB-3 images.
- This function is used for commissioning. In actual production, you are advised to call APIs. You can select **Access Authenticated Using a Token**, **Access Authenticated Using an AK/SK**, or **Access Authenticated Using an Application** based on the authentication mode.

Input Parameters

After a service is deployed, obtain the input parameters of the service on the **Usage Guides** tab page of the service details page.

Figure 3-5 Viewing the Usage Guides tab page
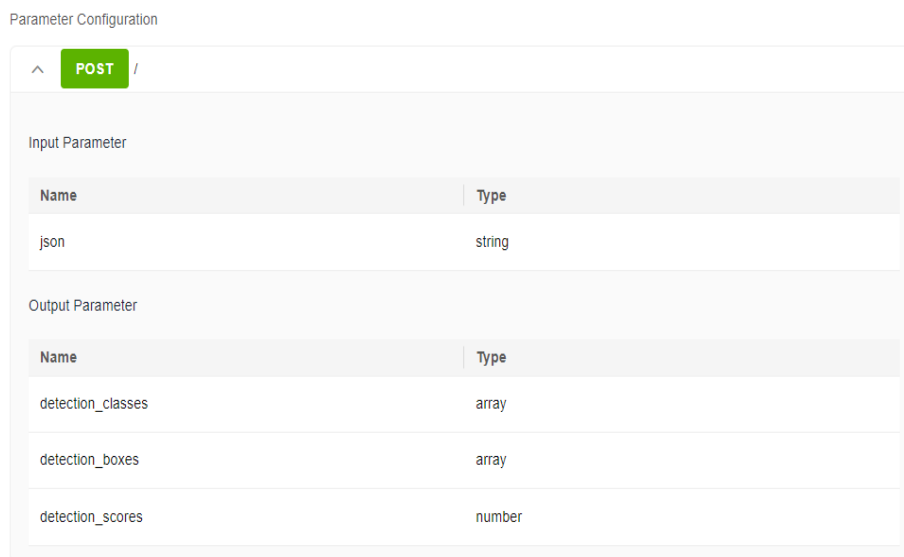
The input parameters displayed on the **Usage Guides** tab page vary depending on the AI application source that you select.

- If your metamodel comes from ExeML or a built-in algorithm, the input and output parameters are defined by ModelArts. For details, see the **Usage Guides** tab page. On the **Prediction** tab page, enter the corresponding JSON text or file for service testing.

- If you use a custom meta model with the inference code and configuration file compiled by yourself (**Specifications for Writing the Model Configuration File**), ModelArts only visualizes your data on the **Usage Guides** tab page. The following figure shows the mapping between the input parameters displayed on the **Usage Guides** tab page and the configuration file.

Figure 3-6 Mapping between the configuration file and Usage Guides

- If your meta model is imported using a model template, the input and output parameters vary with the template. For details, see the description in **Introduction to Model Templates**.

JSON Text Prediction

1. Log in to the ModelArts management console and choose **Service Deployment** > **Real-Time Services**.

2. On the **Real-Time Services** page, click the name of the target service. The service details page is displayed. Enter the inference code on the **Prediction** tab, and click **Predict** to perform prediction.

File Prediction

1. Log in to the ModelArts management console and choose **Service Deployment** > **Real-Time Services**.

2. On the **Real-Time Services** page, click the name of the target service. The service details page is displayed. On the **Prediction** tab page, click **Upload** and select a test file. After the file is uploaded successfully, click **Predict** to perform a prediction test. In **Figure 3-7**, the label, position coordinates, and confidence score are displayed.

Figure 3-7 Image prediction
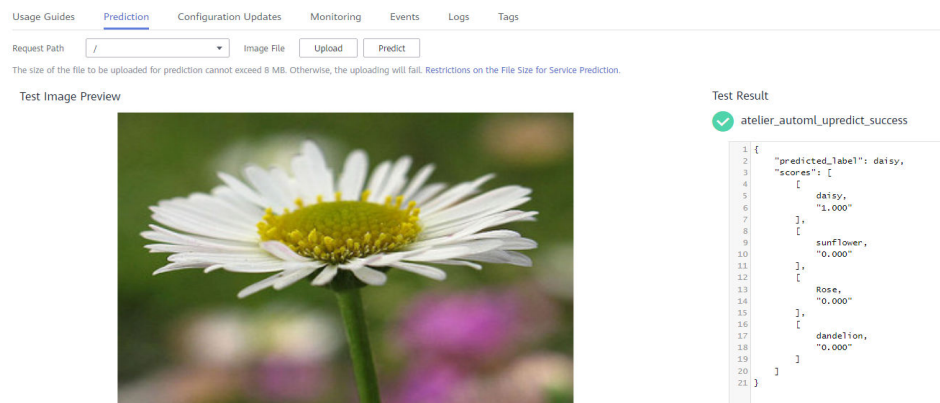
3.1.4 Accessing Real-Time Services

3.1.4.1 Accessing a Real-Time Service

If a real-time service is in the **Running** status, the real-time service has been deployed successfully. This service provides a standard RESTful API for you to call. Before integrating the API to the production environment, commission the API.

By default, APIs of real-time services are accessed using HTTPS. WebSocket-based access is also supported. If you select **WebSocket** during real-time service deployment, the API URL is a WebSocket address after the service is deployed. For details, see **Accessing a Real-Time Service Through WebSocket**.

ModelArts supports the following authentication methods for accessing real-time services (HTTPS requests are used as an example):

- **Access Authenticated Using a Token**
- **Access Authenticated Using an AK/SK**
- **Access Authenticated Using an Application**

ModelArts allows you to call APIs to access real-time services in the following ways:

- **Accessing a Real-Time Service (Public Network Channel)**
- **Accessing a Real-Time Service (VPC High-Speed Channel)**

When you call an API to access a real-time service, the size of the prediction request body and the prediction time are subject to the following limitations:

- The size of a request body cannot exceed 12 MB. Otherwise, the request will fail.
- Due to the limitation of API Gateway, the prediction duration of each request does not exceed 40 seconds.

3.1.4.2 Authentication Mode

3.1.4.2.1 Access Authenticated Using a Token

If a real-time service is in the **Running** state, it has been deployed successfully. This service provides a standard RESTful API for users to call. Before integrating the API to the production environment, commission the API. You can use the following methods to send an inference request to the real-time service:

- **Method 1: Use GUI-based Software for Inference (Postman)**. (Postman is recommended for Windows.)
- **Method 2: Run the cURL Command to Send an Inference Request**. (curl commands are recommended for Linux.)
- **Method 3: Use Python to Send an Inference Request**.
- **Method 4: Use Java to Send an Inference Request**.

Prerequisites

You have obtained a user token, local path to the inference file, URL of the real-time service, and input parameters of the real-time service.

- For details about how to obtain a user token, see **Token-based Authentication**. The real-time service APIs generated by ModelArts do not support tokens whose scope is domain. Therefore, you need to obtain the token whose scope is project.
- The local path to the inference file can be an absolute path (for example, **D:/ test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).
- You can obtain the service URL and input parameters of a real-time service on the Usage Guides tab page of its service details page.

  The API URL is the service URL of the real-time service. If a path is defined for **apis** in the model configuration file, the URL must be followed by the user-defined path, for example, **{***URL of the real-time service***}/predictions/poetry**.

Figure 3-8 Obtaining the API URL and file prediction input parameters of a real-time service
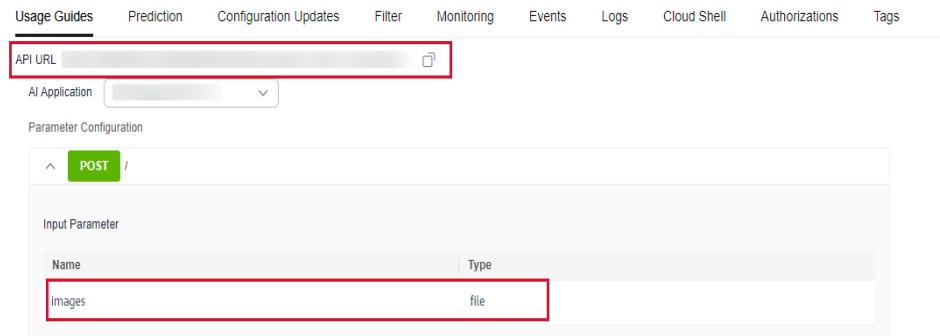
Figure 3-9 Obtaining the API URL and text prediction input parameters of a real-time service
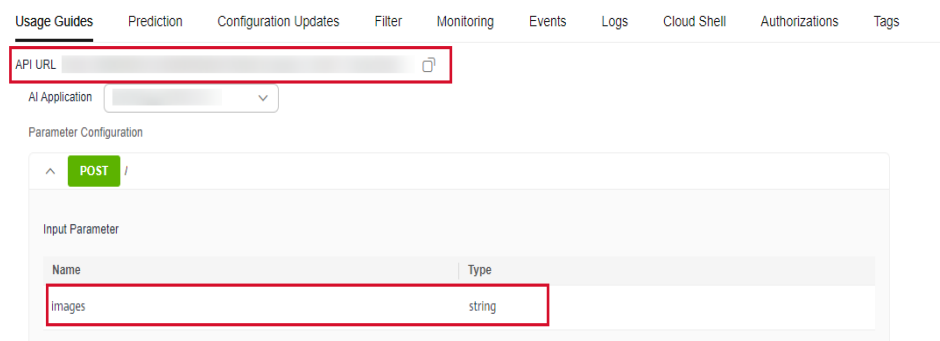
Method 1: Use GUI-based Software for Inference (Postman)

1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 7.24.0 is recommended.

2. Open Postman. **Figure 3-10** shows the Postman interface.

Figure 3-10 Postman interface
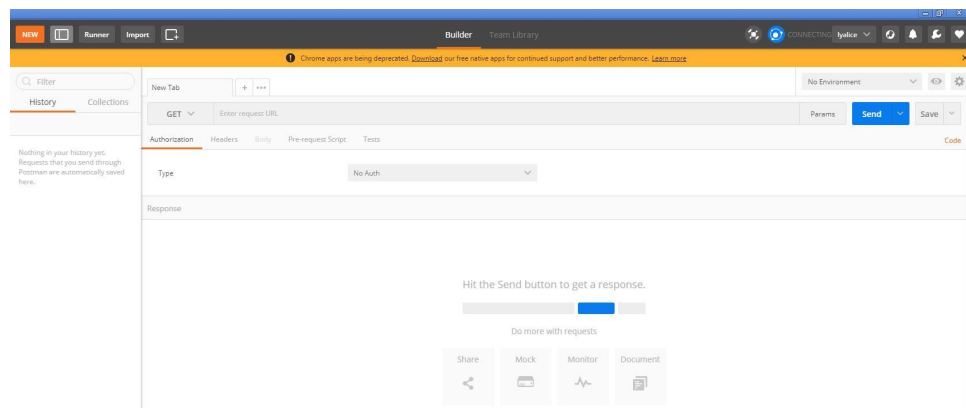
3. Set parameters on Postman. The following uses image classification as an example.

– Select a POST task and copy the API URL to the POST text box. On the **Headers** tab page, set **Key** to **X-Auth-Token** and **Value** to the user token.

📖 NOTE

You can also use the AK and SK to encrypt API calling requests. For details, see **Overview of Session Authentication**.

Figure 3-11 Parameter settings
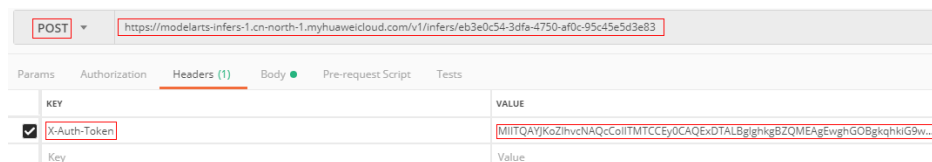
– On the **Body** tab page, file input and text input are available.

- **File input**

  Select **form-data**. Set **KEY** to the input parameter of the AI application, which must be the same as the input parameter of the real-time service. In this example, the **KEY** is **images**. Set **VALUE** to an image to be inferred (only one image can be inferred). See **Figure 3-12**.

  Figure 3-12 Setting parameters on the **Body** tab page
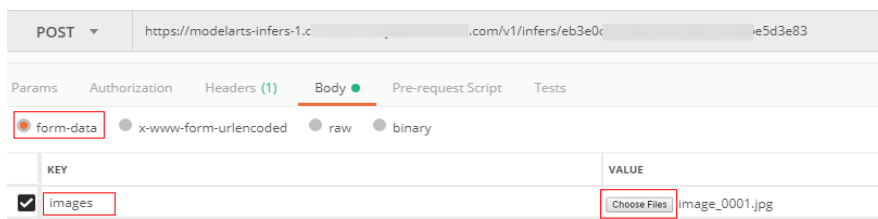
- **Text input**

  Select **raw** and then **JSON(application/json)**. Enter the request body in the text box below. An example request body is as follows:

  ```
  {
    "meta": {
      "uuid": "10eb0091-887f-4839-9929-cbc884f1e20e"
    },
    "data": {
      "req_data": [
        {
          "sepal_length": 3,
          "sepal_width": 1,
          "petal_length": 2.2,
          "petal_width": 4
        }
      ]
    }
  }
  ```

  meta can carry a universally unique identifier (UUID). When the inference result is returned after API calling, the UUID is returned to trace the request. If you do not need this function, leave **meta** blank. **data** contains a **req_data** array for one or multiple pieces of input data. The parameters of each piece of data, such as **sepal_length**

and **sepal_width** in this example are determined by the AI application.

4. After setting the parameters, click **send** to send the request. The result will be displayed in **Response**.

   – Inference result using file input: **Figure 3-13** shows an example. The field values in the return result vary with the AI application.

   – Inference result using text input: **Figure 3-14** shows an example. The request body contains **meta** and **data**. If the request contains **uuid**, **uuid** will be returned in the response. Otherwise, **uuid** is left blank. **data** contains a **resp_data** array for the inference results of one or multiple pieces of input data. The parameters of each result are determined by the AI application, for example, **sepal_length** and **predictresult** in this example.

Figure 3-13 File inference result

Figure 3-14 Text inference result
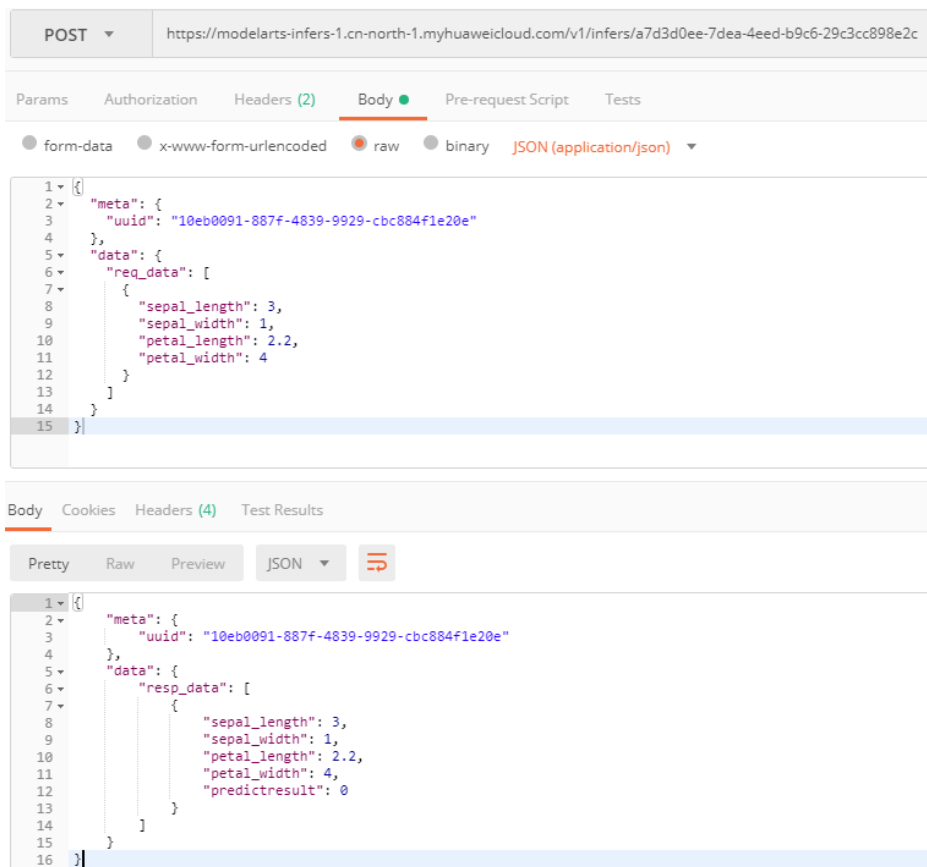
Method 2: Run the cURL Command to Send an Inference Request

The command for sending inference requests can be input as a file or text.

- File input

  ```
  curl -kv -F 'images=@Image path' -H 'X-Auth-Token:Token value' -X POST Real-time service URL
  ```

  - **-k** indicates that SSL websites can be accessed without using a security certificate.

  - **-F** indicates file input. In this example, the parameter name is **images**, which can be changed as required. The image storage path follows **@**.

  - **-H** indicates the header of a POST command. **X-Auth-Token** is the header key, which is fixed. *Token value* indicates the user token.

  - **POST** is followed by the API URL of the real-time service.

  The following is an example of the cURL command for inference with file input:

  ```
  curl -kv -F 'images=@/home/data/test.png' -H 'X-Auth-Token:MIISkAY***80T9wHQ==' -X POST https://
  modelarts-infers-1.xxx/v1/infers/eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
  ```

- Text input

  ```
  curl -kv -d '{"data":{"req_data":
  [{"sepal_length":3,"sepal_width":1,"petal_length":2.2,"petal_width":4}]}}' -H 'X-Auth-
  Token:MIISkAY***80T9wHQ==' -H 'Content-type: application/json' -X POST https://modelarts-
  infers-1.xxx/v1/infers/eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
  ```

  -d indicates the text input of the request body.

Method 3: Use Python to Send an Inference Request

1. Download the Python SDK and configure it in the development tool. For details, see **Integrating the Python SDK for API request signing**.

2. Create a request body for inference.

    – **File input**

    ```
    # coding=utf-8

    import requests

    if __name__ == '__main__':
        # Config url, token and file path.
        url = "URL of the real-time service"
        token = "User token"
        file_path = "Local path to the inference file"

        # Send request.
        headers = {
            'X-Auth-Token': token
        }
        files = {
            'images': open(file_path, 'rb')
        }
        resp = requests.post(url, headers=headers, files=files)

        # Print result.
        print(resp.status_code)
        print(resp.text)
    ```

    The **files** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the file type. The input parameter **images** obtained in **Prerequisites** is an example.

    – **Text input (JSON)**

    The following is an example of the request body for reading the local inference file and performing Base64 encoding:

    ```
    # coding=utf-8

    import base64
    import requests

    if __name__ == '__main__':
        # Config url, token and file path
        url = "URL of the real-time service"
        token = "User token"
        file_path = "Local path to the inference file"
        with open(file_path, "rb") as file:
            base64_data = base64.b64encode(file.read()).decode("utf-8")

        # Set body,then send request
        headers = {
            'Content-Type': 'application/json',
            'X-Auth-Token': token
        }
        body = {
            'image': base64_data
        }
        resp = requests.post(url, headers=headers, json=body)

        # Print result
        print(resp.status_code)
        print(resp.text)
    ```

    The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input

parameter of the string type. The input parameter **images** obtained in **Prerequisites** is an example. The value of **base64_data** in **body** is of the string type.

Method 4: Use Java to Send an Inference Request

1. Download the Java SDK and configure it in the development tool. For details, see **Integrating the Java SDK for API request signing**.

2. (Optional) If the input of the inference request is in the file format, the Java project depends on the httpmime module.

   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 3-15** shows a complete Java dependency library.

      You are advised to use httpmime-x.x.x.jar 4.5 or later. Download httpmime-x.x.x.jar from **https://mvnrepository.com/artifact/org.apache.httpcomponents/httpmime**.

      Figure 3-15 Java dependency library

   b. After **httpmime-x.x.x.jar** is added, add httpmime information to the **.classpath** file of the Java project as follows:
      ```
      <?xml version="1.0" encoding="UTF-8"?>
      <classpath>
      <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
      <classpathentry kind="src" path="src"/>
      <classpathentry kind="lib" path="libs/commons-codec-1.11.jar"/>
      <classpathentry kind="lib" path="libs/commons-logging-1.2.jar"/>
      <classpathentry kind="lib" path="libs/httpclient-4.5.13.jar"/>
      <classpathentry kind="lib" path="libs/httpcore-4.4.13.jar"/>
      <classpathentry kind="lib" path="libs/httpmime-x.x.x.jar"/>
      <classpathentry kind="lib" path="libs/java-sdk-core-3.1.2.jar"/>
      <classpathentry kind="lib" path="libs/okhttp-3.14.9.jar"/>
      <classpathentry kind="lib" path="libs/okio-1.17.2.jar"/>
      <classpathentry kind="output" path="bin"/>
      </classpath>
      ```

3. Create a Java request body for inference.

   – **File input**

      A sample Java request body is as follows:
      ```
      // Package name of the demo.
      package com.apig.sdk.demo;

      import org.apache.http.Consts;
      ```

```java
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.ContentType;
import org.apache.http.entity.mime.MultipartEntityBuilder;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

import java.io.File;

public class MyTokenFile {

    public static void main(String[] args) {
        // Config url, token and filePath
        String url = "URL of the real-time service";
        String token = "User token";
        String filePath = "Local path to the inference file";

        try {
            // Create post
            HttpPost httpPost = new HttpPost(url);

            // Add header parameters
            httpPost.setHeader("X-Auth-Token", token);

            // Add a body if you have specified the PUT or POST method. Special characters, such
    as the double quotation mark ("), contained in the body must be escaped.
            File file = new File(filePath);
            HttpEntity entity = MultipartEntityBuilder.create().addBinaryBody("images",
    file).setContentType(ContentType.MULTIPART_FORM_DATA).setCharset(Consts.UTF_8).build();
            httpPost.setEntity(entity);

            // Send post
            CloseableHttpResponse response = HttpClients.createDefault().execute(httpPost);

            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The **addBinaryBody** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the file type. The file **images** obtained in **Prerequisites** is used as an example.

– **Text input (JSON)**

The following is an example of the request body for reading the local inference file and performing Base64 encoding:

```java
// Package name of the demo.
package com.apig.sdk.demo;

import org.apache.http.HttpHeaders;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

public class MyTokenTest {

    public static void main(String[] args) {
        // Config url, token and body
        String url = "URL of the real-time service";
        String token = "User token";
```

```
        String body = "{}";

        try {
            // Create post
            HttpPost httpPost = new HttpPost(url);

            // Add header parameters
            httpPost.setHeader(HttpHeaders.CONTENT_TYPE, "application/json");
            httpPost.setHeader("X-Auth-Token", token);

            // Special characters, such as the double quotation mark ("), contained in the body
must be escaped.
            httpPost.setEntity(new StringEntity(body));

            // Send post.
            CloseableHttpResponse response = HttpClients.createDefault().execute(httpPost);

            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

body is determined by the text format. JSON is used as an example.

3.1.4.2.2 Access Authenticated Using an AK/SK

If a real-time service is in the **Running** state, the real-time service has been deployed successfully. This service provides a standard RESTful API for users to call. Users can call the API using AK/SK-based authentication.

When AK/SK-based authentication is used, you can use the APIG SDK or ModelArts SDK to access APIs. For details, see **Overview of Session Authentication**. This section describes how to use the APIG SDK to access a real-time service. The process is as follows:

1. **Obtaining an AK/SK Pair**

2. **Obtaining Information About a Real-Time Service**

3. Sending an Inference Request

   – **Method 1: Use Python to Send an Inference Request**

   – **Method 2: Use Java to Send an Inference Request**

> ☐ **NOTE**
>
> 1. AK/SK-based authentication supports API requests with a body not larger than 12 MB. For API requests with a larger body, token-based authentication is recommended.
>
> 2. The local time on the client must be synchronized with the clock server to avoid a large offset in the value of the **X-Sdk-Date** request header. API Gateway checks the time format and compares the time with the time when API Gateway receives the request. If the time difference exceeds 15 minutes, API Gateway will reject the request.

Obtaining an AK/SK Pair

If an AK/SK pair is already available, skip this step. Find the downloaded AK/SK file, which is usually named **credentials.csv**.

As shown in the following figure, the file contains the username, AK, and SK.

Figure 3-16 Content of the credential.csv file

Perform the following operations to generate an AK/SK pair:

1. Register with and log in to the management console.

2. Click the username and choose **My Credentials** from the drop-down list.

3. On the **My Credentials** page, choose **Access Keys** in the navigation pane.

4. Click **Create Access Key**. The **Identity Verification** dialog box is displayed.

5. Complete the identity authentication as prompted, download the access key, and keep it secure.

Obtaining Information About a Real-Time Service

When calling an API, you need to obtain the API address and input parameters of the real-time service. The procedure is as follows:

1. Log in to the ModelArts management console. In the left navigation pane, choose **Service Deployment** > **Real-Time Services**. By default, the system switches to the **Real-Time Services** page.

2. Click the name of the target service. The service details page is displayed.

3. On the details page of a real-time service, obtain the API address and input parameters of the service.

   The API URL is the service URL of the real-time service. If a path is defined for **apis** in the model configuration file, the URL must be followed by the user-defined path, for example, **{***URL of the real-time service***}/predictions/poetry**.

   Figure 3-17 Obtaining the API URL and file prediction input parameters of a real-time service
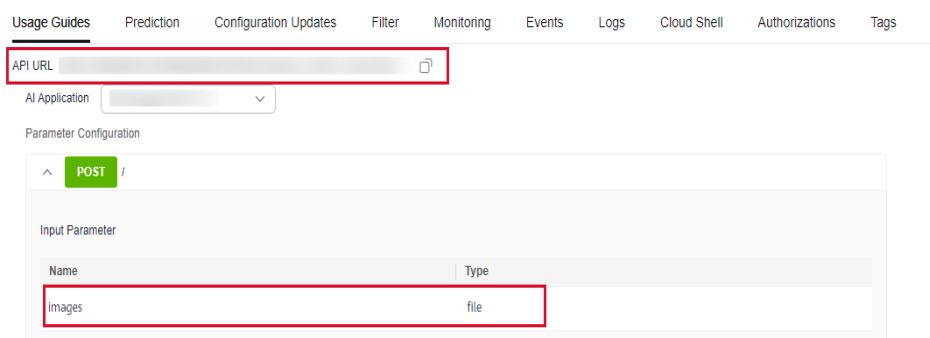
Figure 3-18 Obtaining the API URL and text prediction input parameters of a real-time service

Method 1: Use Python to Send an Inference Request

1. Download the Python SDK and configure it in the development tool. For details, see **Integrating the Python SDK for API request signing**.

2. Create a request body for inference.

   – **File input**

   ```
   # coding=utf-8

   import requests
   import os
   from apig_sdk import signer

   if __name__ == '__main__':
       # Config url, ak, sk and file path.
       url = "URL of the real-time service"
       # Hardcoded or plaintext AK/SK is risky. For security, encrypt your AK/SK and store
   them in the configuration file or environment variables.
       # In this example, the AK/SK are stored in environment variables for identity
   authentication. Before running this example, set environment variables
   HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK.
       ak = os.environ["HUAWEICLOUD_SDK_AK"]
       sk = os.environ["HUAWEICLOUD_SDK_SK"]
       file_path = "Local path to the inference file"

       # Create request, set method, url, headers and body.
       method = 'POST'
       headers = {"x-sdk-content-sha256": "UNSIGNED-PAYLOAD"}
       request = signer.HttpRequest(method, url, headers)

       # Create sign, set the AK/SK to sign and authenticate the request.
       sig = signer.Signer()
       sig.Key = ak
       sig.Secret = sk
       sig.Sign(request)

       # Send request
       files = {'images': open(file_path, 'rb')}
       resp = requests.request(request.method, request.scheme + "://" + request.host + request.uri,
   headers=request.headers, files=files)

       # Print result
       print(resp.status_code)
       print(resp.text)
   ```

   file_path is the local path to the inference file. The path can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/ test.png** for Linux) or a relative path (for example, **./test.png**).

Request body format of **files**: files = {"Request parameter": ("Load path", File content, "File type")}. For details about parameters of **files**, see **Table 3-9**.

Table 3-9 Parameters of **files**

Parameter	Mandatory	Description
Request parameter	Yes	Enter the parameter name of the real-time service.
Load path	No	Path in which the file is stored.
File content	Yes	Content of the file to be uploaded.
File type	No	Type of the file to be uploaded, which can be one of the following options: • **txt**: text/plain • **jpg/jpeg**: image/jpeg • **png**: image/png

– **Text input (JSON)**

The following is an example of the request body for reading the local inference file and performing Base64 encoding:

```
# coding=utf-8

import base64
import json
import os
import requests
from apig_sdk import signer

if __name__ == '__main__':
    # Config url, ak, sk and file path.
    url = "URL of the real-time service"
    # Hardcoded or plaintext AK/SK is risky. For security, encrypt your AK/SK and store
them in the configuration file or environment variables.
    # In this example, the AK/SK are stored in environment variables for identity
authentication. Before running this example, set environment variables
HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK.
    ak = os.environ["HUAWEICLOUD_SDK_AK"]
    sk = os.environ["HUAWEICLOUD_SDK_SK"]
    file_path = "Local path to the inference file"
    with open(file_path, "rb") as file:
        base64_data = base64.b64encode(file.read()).decode("utf-8")

    # Create request, set method, url, headers and body.
    method = 'POST'
    headers = {
        'Content-Type': 'application/json'
    }
    body = {
        'image': base64_data
    }
    request = signer.HttpRequest(method, url, headers, json.dumps(body))

    # Create sign, set the AK/SK to sign and authenticate the request.
```

```
sig = signer.Signer()
sig.Key = ak
sig.Secret = sk
sig.Sign(request)

# Send request
resp = requests.request(request.method, request.scheme + "://" + request.host + request.uri,
headers=request.headers, data=request.body)

# Print result
print(resp.status_code)
print(resp.text)
```

The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. **image** is used as an example. The value of **base64_data** in **body** is of the string type.

Method 2: Use Java to Send an Inference Request

1. Download the Java SDK and configure it in the development tool.

2. Create a Java request body for inference.

   In the APIG Java SDK, **request.setBody()** can only be a string. Therefore, only text inference requests are supported. If a file is input, convert the file into text using Base64.

   – **File input**

     The following is an example of the request body (JSON) for reading the local inference file and performing Base64 encoding.

```
package com.apig.sdk.demo;
import com.cloud.apigateway.sdk.utils.Client;
import com.cloud.apigateway.sdk.utils.Request;
import org.apache.commons.codec.binary.Base64;
import org.apache.http.HttpHeaders;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpRequestBase;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
public class MyAkSkTest2 {
    public static void main(String[] args) {
        String url = "URL of the real-time service";
        // Hard-coded or plaintext AK/SK is risky. For security, encrypt your AK/SK and store
them in the configuration file or environment variables.
        // In this example, the AK/SK are stored in environment variables for identity
authentication. Before running this example, set environment variables
HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK.
        String ak = System.getenv("HUAWEICLOUD_SDK_AK");
        String sk = System.getenv("HUAWEICLOUD_SDK_SK");
        String filePath = "Local path to the inference file";
        try {
            // Create request
            Request request = new Request();
            // Set the AK/SK to sign and authenticate the request.
            request.setKey(ak);
            request.setSecret(sk);
            // Specify a request method, such as GET, PUT, POST, DELETE, HEAD, and PATCH.
            request.setMethod(HttpPost.METHOD_NAME);
            // Add header parameters
            request.addHeader(HttpHeaders.CONTENT_TYPE, "application/json");
            // Set a request URL in the format of https://{Endpoint}/{URI}.
            request.setUrl(url);
```

```
            // build your json body
            String body = "{\"image\":\"" + getBase64FromFile(filePath) + "\"}";
            // Special characters, such as the double quotation mark ("), contained in the body
    must be escaped.
            request.setBody(body);
            // Sign the request.
            HttpRequestBase signedRequest = Client.sign(request);
            // Send request.
            CloseableHttpResponse response = HttpClients.createDefault().execute(signedRequest);
            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * Convert the file into a byte array and Base64 encode it
     * @return
     */
    private static String getBase64FromFile(String filePath) {
        // Convert the file into a byte array
        InputStream in = null;
        byte[] data = null;
        try {
            in = new FileInputStream(filePath);
            data = new byte[in.available()];
            in.read(data);
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Base64 encode
        return new String(Base64.encodeBase64(data));
    }
}
```

⚠ **CAUTION**

If Base64 encoding is used, you need to add the code for decoding the request body to the model inference code.

- **Text input (JSON)**

```
// Package name of the demo.
package com.apig.sdk.demo;

import com.cloud.apigateway.sdk.utils.Client;
import com.cloud.apigateway.sdk.utils.Request;
import org.apache.http.HttpHeaders;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpRequestBase;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

public class MyAkSkTest {

    public static void main(String[] args) {
        String url = "URL of the real-time service";
        // Hard-coded or plaintext AK/SK is risky. For security, encrypt your AK/SK and store
    them in the configuration file or environment variables.
        // In this example, the AK/SK are stored in environment variables for identity
    authentication. Before running this example, set environment variables
    HUAWEICLOUD_SDK_AK and HUAWEICLOUD_SDK_SK.
        String ak = System.getenv("HUAWEICLOUD_SDK_AK");
        String sk = System.getenv("HUAWEICLOUD_SDK_SK");
```

```
        try {
            // Create request
            Request request = new Request();

            // Set the AK/SK to sign and authenticate the request.
            request.setKey(ak);
            request.setSecret(sk);

            // Specify a request method, such as GET, PUT, POST, DELETE, HEAD, and PATCH.
            request.setMethod(HttpPost.METHOD_NAME);

            // Add header parameters
            request.addHeader(HttpHeaders.CONTENT_TYPE, "application/json");

            // Set a request URL in the format of https://{Endpoint}/{URI}.
            request.setUrl(url);

            // Special characters, such as the double quotation mark ("), contained in the body
must be escaped.
            String body = "{}";
            request.setBody(body);

            // Sign the request.
            HttpRequestBase signedRequest = Client.sign(request);

            // Send request.
            CloseableHttpResponse response = HttpClients.createDefault().execute(signedRequest);

            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

body is determined by the text format. JSON is used as an example.

3.1.4.2.3 Access Authenticated Using an Application

You can enable application authentication when deploying a real-time service. ModelArts registers an API that supports application authentication for the service. After this API is authorized to an application, you can call this API using the AppKey/AppSecret or AppCode of the application.

The process of application authentication for a real-time service is as follows:

1. **Enabling Application Authentication**: Enable application authentication. You can select an existing application or create an application.

2. **Managing Authorization of Real-Time Services**: Manage the created application, including viewing, resetting, or deleting the application, binding or unbinding real-time services for the application, and obtaining the AppKey/ AppSecret or AppCode.

3. **Application Authentication**: Authentication is required for calling an API that supports application authentication. Two authentication modes (AppKey +AppSecret or AppCode) are provided. You can select either of them.

4. Sending an Inference Request

    – **Method 1: Use Python to Send an Inference request Through AppKey/ AppSecret-based Authentication**

    – **Method 2: Use Java to Send an Inference request Through AppKey/ AppSecret-based Authentication**

– **Method 3: Use Python to Send an Inference request Through AppCode-based Authentication**

– **Method 4: Use Java to Send an Inference request Through AppCode-based Authentication**

Prerequisites

- Data has been prepared. Specifically, you have created an AI application in the **Normal** state in ModelArts.

- The account is not in arrears to ensure available resources for service running.

- The local path to the inference file has been obtained. The path can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).

Enabling Application Authentication

When deploying an AI application as a real-time service, you can enable application authentication. You can also modify a deployed real-time service to support application authentication.

1. Log in to the ModelArts management console and choose **Service Deployment > Real-Time Services**.

2. Enable application authentication.

   – When deploying a model as a real-time service, configure the required parameters and enable application authentication on the **Deploy** page.

   – For a deployed real-time service, go to the **Real-Time Services** page, and click **Modify** in the **Operation** column of the service. On the service modification page that is displayed, enable application authentication.

   Figure 3-19 Enabling application authentication
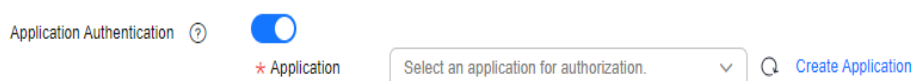
3. Select an application for authorization from the drop-down list. If no application is available, create one as follows:

   – Click **Create Application** on the right, enter the application name and description, and click **OK**. By default, the application name starts with **app_**. You can change the application name.

   – On the **Service Deployment** > **Real-Time Services** page, click **Authorize**. On the **Manage Authorization of Real-Time Services** page, click **Create Application**. For details, see **Managing Authorization of Real-Time Services**.

4. After enabling application authentication, authorize a service that supports application authentication to the application. Then, you can use the generated AppKey/AppSecret or AppCode to call the service's API that supports application authentication.

Managing Authorization of Real-Time Services

If you want to use application authentication, it is a good practice to create an application on the authorization management page before deploying a real-time

service. In the navigation pane, choose **Service Deployment** > **Real-Time Services**. On the **Real-Time Services** page, click **Authorize**. The **Manage Authorization of Real-Time Services** dialog box is displayed. From there, you can create and manage applications, including viewing, resetting, and deleting applications, unbinding real-time services from applications, and obtaining the AppKey/AppSecret or AppCode.

Figure 3-20 Managing authorization for real-time services
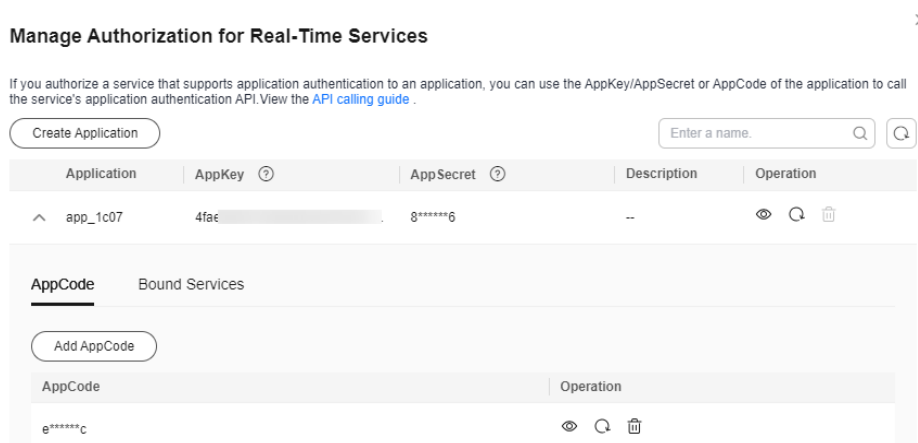
- **Creating an application**

  Click **Create Application**, enter the application name and description, and click **OK**. By default, the application name starts with **app_**. You can change the application name.

- **Viewing, resetting, or deleting an application**

  View, reset, or delete an application by clicking the corresponding icon in the **Operation** column of the application. After an application is created, the AppKey and AppSecret are automatically generated for application authentication.

- **Unbinding a service**

  In front of the target application name, click ⌄ to view the real-time services bound to the application. Click **Unbind** in the **Operation** column to cancel the binding. Then, this API cannot be called.

- **Obtaining the AppKey/AppSecret or AppCode**

  Application authentication is required for API calling. The AppKey and AppSecret are automatically generated during application creation. Click ⊙ in the **Operation** column of the application in the **Manage Authorization of Real-Time Services** dialog box to view the complete AppSecret. Click ⌄ in front of the application name to expand the drop-down list. Click **+Add AppCode** to automatically generate an AppCode. Then, click ⊙ in the **Operation** column to view the complete AppCode.

Figure 3-21 Adding the AppCode
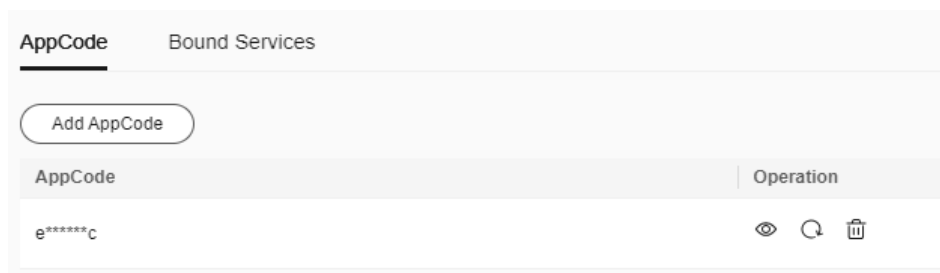
Application Authentication

When a real-time service that supports application authentication is in the **Running** state, the service' API can be called. Before calling the API, perform application authentication.

When application authentication is used and the simplified authentication mode is enabled, you can use the AppKey/AppSecret for signature and verification, or AppCode for simplified authentication of API requests. (Simplified authentication is used by ModelArts by default.) AppKey/AppSecret-based authentication is recommended because it is more secure than AppCode-based authentication.

- **AppKey/AppSecret-based authentication**: The AppKey and AppSecret are used to encrypt a request, identify the sender, and prevent the request from being modified. When using AppKey/AppSecret-based authentication, use a dedicated signing SDK to sign requests.

  - AppKey: access key ID of the application, which is a unique identifier used together with a secret access key to sign requests cryptographically.

  - AppSecret: application secret access key, used together with the Access Key ID to encrypt the request, identify the sender, and prevent the request from being tempered.

  AppKeys can be used for simplified authentication. When an API is called, the **apikey** parameter (value: **AppKey**) is added to the HTTP request header to accelerate authentication.

- **AppCode-based authentication**: Requests are authenticated using AppCodes.

  In AppCode-based authentication, the **X-Apig-AppCode** parameter (value: **AppCode**) is added to the HTTP request header when an API is called. The request content does not need to be signed. The API gateway only verifies the AppCode, achieving quick response.

You can obtain the API, AppKey/AppSecret, and AppCode from the **Usage Guides** tab page on the service details page (see **Figure 3-22**) or from the real-time service authorization management page (see **Figure 3-20**). Use the API URL in the red box in the following figure. If a path is defined for **apis** in the model configuration file, the URL must be followed by the user-defined path, for example, **{***URL of the real-time service***}/predictions/poetry**.

Figure 3-22 Obtaining the API address

Method 1: Use Python to Send an Inference request Through AppKey/AppSecret-based Authentication

1. Download the Python SDK and configure it in the development tool.

2. Create a request body for inference.

   - **File input**

```
# coding=utf-8

import requests
import os
from apig_sdk import signer

if __name__ == '__main__':
    # Config url, ak, sk and file path.
    url = "URL of the real-time service"
    # Hardcoded or plaintext app_key and app_secret are risky. For security, encrypt and store them in the configuration file or environment variables.
    # In this example, the app_key and app_secret are stored in environment variables for identity authentication. Before running this example, set environment variables HUAWEICLOUD_APP_KEY and HUAWEICLOUD_APP_SECRET.
    app_key = os.environ["HUAWEICLOUD_APP_KEY"]
    app_secret= os.environ["HUAWEICLOUD_APP_SECRET"]
    file_path = "Local path to the inference file"

    # Create request, set method, url, headers and body.
    method = 'POST'
    headers = {"x-sdk-content-sha256": "UNSIGNED-PAYLOAD"}
    request = signer.HttpRequest(method, url, headers)

    # Create sign, set the AK/SK to sign and authenticate the request.
    sig = signer.Signer()
    sig.Key = app_key
    sig.Secret = app_secret
    sig.Sign(request)

    # Send request
    files = {'images': open(file_path, 'rb')}
    resp = requests.request(request.method, request.scheme + "://" + request.host + request.uri, headers=request.headers, files=files)

    # Print result
    print(resp.status_code)
    print(resp.text)
```

   Request body format of **files**: files = {"Request parameter": ("Load path", File content, "File type")}. For details about parameters of **files**, see **Table 3-10**.

   Table 3-10 Parameters of **files**

Parameter	Mandatory	Description
Request parameter	Yes	Parameter name of the real-time service.
Load path	No	Path in which the file is stored.
File content	Yes	Content of the file to be uploaded.

Paramete r	Mand atory	Description
File type	No	Type of the file to be uploaded, which can be one of the following options: ● **txt**: text/plain ● **jpg/jpeg**: image/jpeg ● **png**: image/png

– **Text input (JSON)**

The following is an example of the request body for reading the local inference file and performing Base64 encoding:

```
# coding=utf-8

import base64
import json
import os
import requests
from apig_sdk import signer

if __name__ == '__main__':
    # Config url, ak, sk and file path.
    url = "URL of the real-time service"
    # Hardcoded or plaintext app_key and app_secret are risky. For security, encrypt and store them in the configuration file or environment variables.
    # In this example, the app_key and app_secret are stored in environment variables for identity authentication. Before running this example, set environment variables HUAWEICLOUD_APP_KEY and HUAWEICLOUD_APP_SECRET.
    app_key = os.environ["HUAWEICLOUD_APP_KEY"]
    app_secret= os.environ["HUAWEICLOUD_APP_SECRET"]
    file_path = "Local path to the inference file"
    with open(file_path, "rb") as file:
        base64_data = base64.b64encode(file.read()).decode("utf-8")

    # Create request, set method, url, headers and body.
    method = 'POST'
    headers = {
        'Content-Type': 'application/json'
    }
    body = {
        'image': base64_data
    }
    request = signer.HttpRequest(method, url, headers, json.dumps(body))

    # Create sign, set the AppKey&AppSecret to sign and authenticate the request.
    sig = signer.Signer()
    sig.Key = app_key
    sig.Secret = app_secret
    sig.Sign(request)

    # Send request
    resp = requests.request(request.method, request.scheme + "://" + request.host + request.uri, headers=request.headers, data=request.body)

    # Print result
    print(resp.status_code)
    print(resp.text)
```

The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. **image** is used as an example. The value of **base64_data** in **body** is of the string type.

Method 2: Use Java to Send an Inference request Through AppKey/ AppSecret-based Authentication

1. Download the Java SDK and configure it in the development tool.
2. Create a Java request body for inference.

   In the APIG Java SDK, **request.setBody()** can only be a string. Therefore, only text inference requests are supported.

   The following is an example of the request body (JSON) for reading the local inference file and performing Base64 encoding:

```java
// Package name of the demo.
package com.apig.sdk.demo;

import com.cloud.apigateway.sdk.utils.Client;
import com.cloud.apigateway.sdk.utils.Request;
import org.apache.http.HttpHeaders;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpRequestBase;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

public class MyAkSkTest {

    public static void main(String[] args) {
        String url = "URL of the real-time service";
        // Hard-coded or plaintext app_key and app_secret are risky. For security, encrypt and store
them in the configuration file or environment variables.
        // In this example, the app_key and app_secret are stored in environment variables for
identity authentication. Before running this example, set environment variables
HUAWEICLOUD_APP_KEY and HUAWEICLOUD_APP_SECRET.
        String appKey = System.getenv("HUAWEICLOUD_APP_KEY");
        String appSecret = System.getenv("HUAWEICLOUD_APP_SECRET");
        String body = "{}";

        try {
            // Create request
            Request request = new Request();

            // Set the AK/AppSecret to sign and authenticate the request.
            request.setKey(appKey);
            request.setSecret(appSecret);

            // Specify a request method, such as GET, PUT, POST, DELETE, HEAD, and PATCH.
            request.setMethod(HttpPost.METHOD_NAME);

            // Add header parameters
            request.addHeader(HttpHeaders.CONTENT_TYPE, "application/json");

            // Set a request URL in the format of https://{Endpoint}/{URI}.
            request.setUrl(url);

            // Special characters, such as the double quotation mark ("), contained in the body must be
escaped.
            request.setBody(body);

            // Sign the request.
            HttpRequestBase signedRequest = Client.sign(request);

            // Send request.
            CloseableHttpResponse response = HttpClients.createDefault().execute(signedRequest);

            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
```

```
            }
        }
    }
```

body is determined by the text format. JSON is used as an example.

Method 3: Use Python to Send an Inference request Through AppCode-based Authentication

1. Download the Python SDK and configure it in the development tool.

2. Create a request body for inference.

   – **File input**

   ```
   # coding=utf-8

   import requests
   import os

   if __name__ == '__main__':
       # Config url, app code and file path.
       url = "URL of the real-time service"
       # Hardcoded or plaintext app_code is risky. For security, encrypt and store it in the
   configuration file or environment variables.
       # In this example, the app_code is stored in environment variables for identity
   authentication. Before running this example, set environment variable
   HUAWEICLOUD_APP_CODE.
       app_code = os.environ["HUAWEICLOUD_APP_CODE"]
       file_path = "Local path to the inference file"

       # Send request.
       headers = {
           'X-Apig-AppCode': app_code
       }
       files = {
           'images': open(file_path, 'rb')
       }
       resp = requests.post(url, headers=headers, files=files)

       # Print result
       print(resp.status_code)
       print(resp.text)
   ```

   The **files** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the file type. In this example, **images** is used.

   – **Text input (JSON)**

   The following is an example of the request body for reading the local inference file and performing Base64 encoding:

   ```
   # coding=utf-8

   import base64
   import requests
   import os

   if __name__ == '__main__':
       # Config url, app code and request body.
       url = "URL of the real-time service"
       # Hardcoded or plaintext app_code is risky. For security, encrypt and store it in the
   configuration file or environment variables.
       # In this example, the app_code is stored in environment variables for identity
   authentication. Before running this example, set environment variable
   HUAWEICLOUD_APP_CODE.
       app_code = os.environ["HUAWEICLOUD_APP_CODE"]
       file_path = "Local path to the inference file"
       with open(file_path, "rb") as file:
           base64_data = base64.b64encode(file.read()).decode("utf-8")
   ```

```
# Send request
headers = {
    'Content-Type': 'application/json',
    'X-Apig-AppCode': app_code
}
body = {
    'image': base64_data
}
resp = requests.post(url, headers=headers, json=body)

# Print result
print(resp.status_code)
print(resp.text)
```

The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. **image** is used as an example. The value of **base64_data** in **body** is of the string type.

Method 4: Use Java to Send an Inference request Through AppCode-based Authentication

1. Download the Java SDK and configure it in the development tool.

2. (Optional) If the input of the inference request is in the file format, the Java project depends on the httpmime module.

   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 3-23** shows a complete Java dependency library.

      You are advised to use httpmime-x.x.x.jar 4.5 or later. Download httpmime-x.x.x.jar from **https://mvnrepository.com/artifact/org.apache.httpcomponents/httpmime**.

   Figure 3-23 Java dependency library

   b. After **httpmime-x.x.x.jar** is added, add httpmime information to the **.classpath** file of the Java project as follows:
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <classpath>
   <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
   <classpathentry kind="src" path="src"/>
   <classpathentry kind="lib" path="libs/commons-codec-1.11.jar"/>
   <classpathentry kind="lib" path="libs/commons-logging-1.2.jar"/>
   <classpathentry kind="lib" path="libs/httpclient-4.5.13.jar"/>
   ```

```
<classpathentry kind="lib" path="libs/httpcore-4.4.13.jar"/>
<classpathentry kind="lib" path="libs/httpmime-x.x.x.jar"/>
<classpathentry kind="lib" path="libs/java-sdk-core-3.1.2.jar"/>
<classpathentry kind="lib" path="libs/okhttp-3.14.9.jar"/>
<classpathentry kind="lib" path="libs/okio-1.17.2.jar"/>
<classpathentry kind="output" path="bin"/>
</classpath>
```

3. Create a Java request body for inference.

   – **File input**

   A sample Java request body is as follows:

```
// Package name of the demo.
package com.apig.sdk.demo;

import org.apache.http.Consts;
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.ContentType;
import org.apache.http.entity.mime.MultipartEntityBuilder;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

import java.io.File;

public class MyAppCodeFile {

    public static void main(String[] args) {
        String url = "URL of the real-time service";
        // Hard-coded or plaintext appCode is risky. For security, encrypt and store it in the
configuration file or environment variables.
        // In this example, the appCode is stored in environment variables for identity
authentication. Before running this example, set environment variable
HUAWEICLOUD_APP_CODE.
        String appCode = System.getenv("HUAWEICLOUD_APP_CODE");
        String filePath = "Local path to the inference file";

        try {
            // Create post
            HttpPost httpPost = new HttpPost(url);

            // Add header parameters
            httpPost.setHeader("X-Apig-AppCode", appCode);

            // Special characters, such as the double quotation mark ("), contained in the body
must be escaped.
            File file = new File(filePath);
            HttpEntity entity = MultipartEntityBuilder.create().addBinaryBody("images",
file).setContentType(ContentType.MULTIPART_FORM_DATA).setCharset(Consts.UTF_8).build();
            httpPost.setEntity(entity);

            // Send post
            CloseableHttpResponse response = HttpClients.createDefault().execute(httpPost);

            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
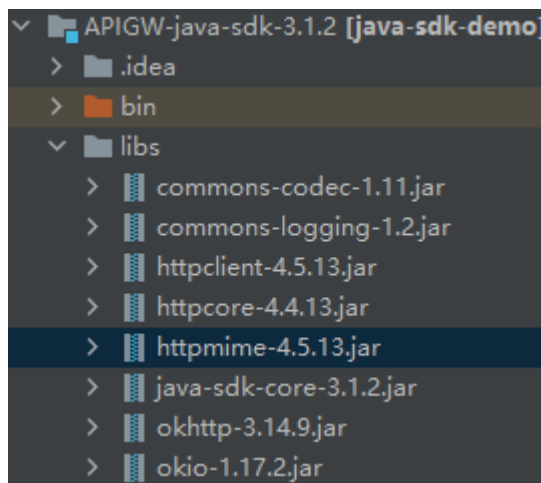
   The **addBinaryBody** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the file type. In this example, **images** is used.

   – **Text input (JSON)**

The following is an example of the request body for reading the local inference file and performing Base64 encoding:

```java
// Package name of the demo.
package com.apig.sdk.demo;

import org.apache.http.HttpHeaders;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;

public class MyAppCodeTest {

    public static void main(String[] args) {
        String url = "URL of the real-time service";
        // Hard-coded or plaintext appCode is risky. For security, encrypt and store it in the
configuration file or environment variables.
        // In this example, the appCode is stored in environment variables for identity
authentication. Before running this example, set environment variable
HUAWEICLOUD_APP_CODE.
        String appCode = System.getenv("HUAWEICLOUD_APP_CODE");
        String body = "{}";

        try {
            // Create post
            HttpPost httpPost = new HttpPost(url);

            // Add header parameters
            httpPost.setHeader(HttpHeaders.CONTENT_TYPE, "application/json");
            httpPost.setHeader("X-Apig-AppCode", appCode);

            // Special characters, such as the double quotation mark ("), contained in the body
must be escaped.
            httpPost.setEntity(new StringEntity(body));

            // Send post
            CloseableHttpResponse response = HttpClients.createDefault().execute(httpPost);

            // Print result
            System.out.println(response.getStatusLine().getStatusCode());
            System.out.println(EntityUtils.toString(response.getEntity()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

body is determined by the text format. JSON is used as an example.

3.1.4.3 Access Mode

3.1.4.3.1 Accessing a Real-Time Service (Public Network Channel)

Context

By default, ModelArts inference uses the public network to access real-time services. After a real-time service is deployed, a standard RESTful API is provided for you to call. You can view the API URL on the **Usage Guides** tab page of the service details page.

| Usage Guides | Prediction | Configuration Updates | Filter | Monitoring |

API URL
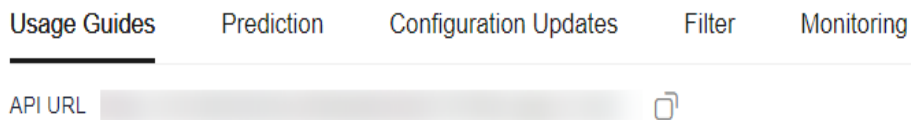
Accessing a Real-Time Service

The following authentication modes are available for accessing real-time services from a public network:

- **Access Authenticated Using a Token**
- **Access Authenticated Using an AK/SK**
- **Access Authenticated Using an Application**

3.1.4.3.2 Accessing a Real-Time Service (VPC High-Speed Channel)

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 3-25 High-speed access through VPC peering
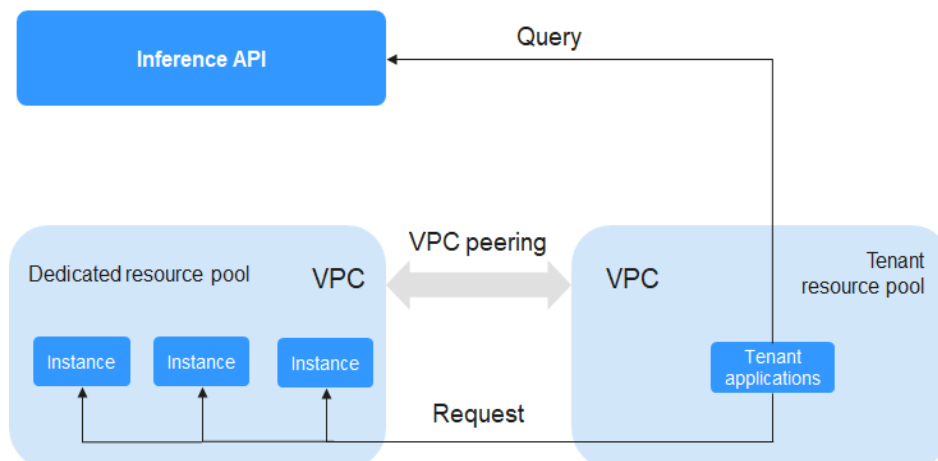
Preparations

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, there is a limit on how often you can get the IP address and port number of a real-time service. 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 real-time 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 3-26 Locating the target dedicated resource pool
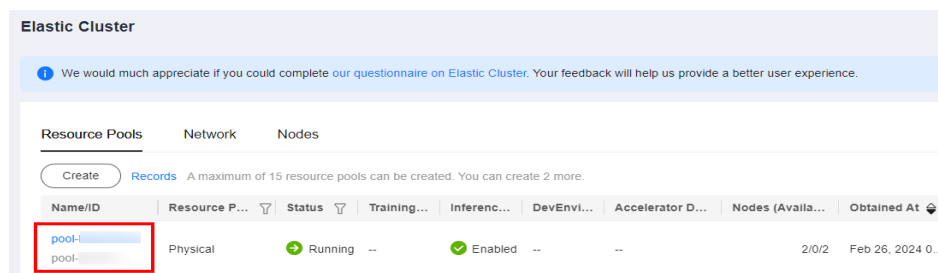
Figure 3-27 Obtaining the network configuration
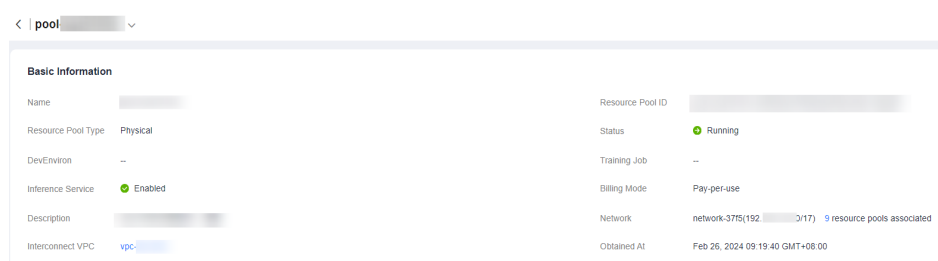
Figure 3-28 Interconnecting the VPC

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 3-29 Selecting a VPC when purchasing an ECS
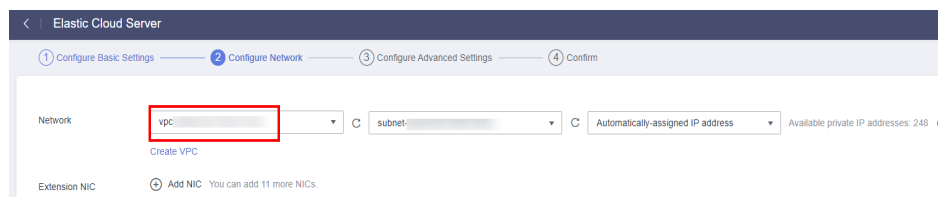
Figure 3-30 Viewing VPC information

Step 3 Obtain the IP address and port number of the real-time 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
```

- Method 1: Obtain the IP address and port number using GUI software.

Figure 3-31 Example response
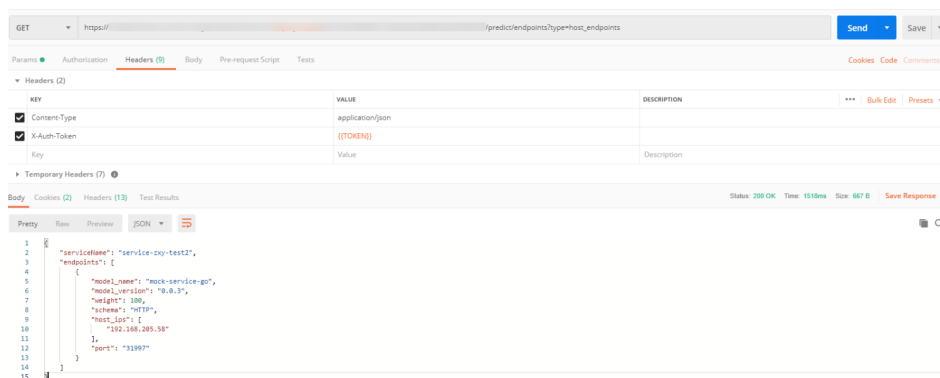
- Method 2: Obtain the IP address and port number using Python.

  The following parameters in the Python code below need to be modified:

  - **project_id**: your project ID. To obtain it, see **Obtaining a Project ID and Name**.

  - **service_id**: service ID, which can be viewed on the service details page.

  - **REGION_ENDPOINT**: service endpoint. To obtain it, see **Endpoint**.

```
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 3-32 Accessing a real-time service

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

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

☐ **NOTE**

High-speed access does not support load balancing. You need to customize load balancing policies when you deploy multiple instances.

----End

3.1.4.4 Accessing a Real-Time Service Through WebSocket

Context

WebSocket is a network transmission protocol that supports full-duplex communication over a single TCP connection. It is located at the application layer in the OSI model. The WebSocket communication protocol was established by IETF as standard RFC 6455 in 2011 and supplemented by RFC 7936. The WebSocket API in the Web IDL is standardized by W3C.

WebSocket simplifies data exchange between the client and server and allows the server to proactively push data to the client. In the WebSocket API, if the initial handshake between the client and server is successful, a persistent connection can be established between them and bidirectional data transmission can be performed.

Prerequisites

- A real-time service has been deployed with **WebSocket** enabled.
- The image for importing the AI application is WebSocket-compliant.

Constraints

- WebSocket supports only the deployment of real-time services.
- WebSocket supports only real-time services deployed using AI applications imported from custom images.

Calling a WebSocket Real-Time Service

WebSocket itself does not require additional authentication. ModelArts WebSocket is WebSocket Secure-compliant, regardless of whether WebSocket or WebSocket Secure is enabled in the custom image. WebSocket Secure supports only one-way authentication, from the client to the server.

You can use one of the following authentication methods provided by ModelArts:

- Access Authenticated Using a Token
- Access Authenticated Using an AK/SK
- Access Authenticated Using an Application

The following section uses GUI software Postman for prediction and token authentication as an example to describe how to call WebSocket.

1. **Establish a WebSocket connection.**
2. **Exchange data between the WebSocket client and the server.**

Step 1 Establish a WebSocket connection.

1. Open Postman of a version later than 8.5, for example, 10.12.0. Click ☰ in the upper left corner and choose **File** > **New**. In the displayed dialog box, select **WebSocket Request** (beta version currently).

Figure 3-33 WebSocket Request
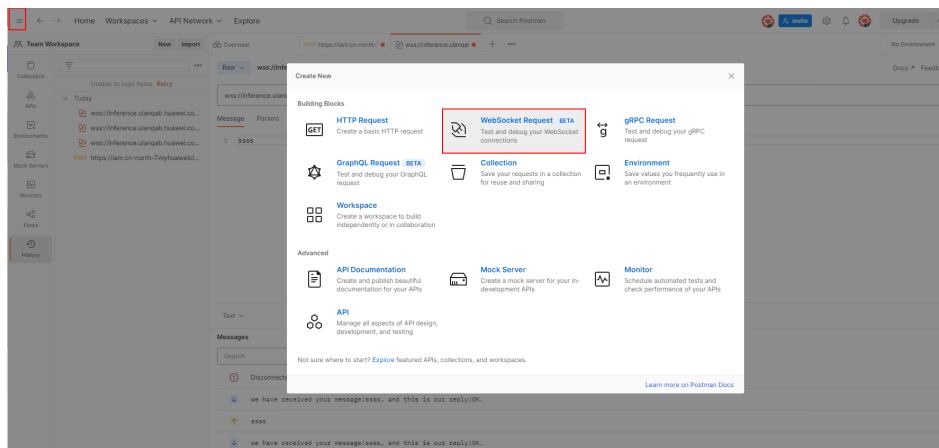
2. Configure parameters for the WebSocket connection.

   Select **Raw** in the upper left corner. Do not select **Socket.IO** (a type of WebSocket implementation, which requires that both the client and the server run on **Socket.IO**). In the address box, enter the **API Address** obtained on the **Usage Guides** tab on the service details page. If there is a finer-grained URL in the custom image, add the URL to the end of the address. If **queryString** is available, add this parameter in the **params** column. Add authentication information into the header. The header varies depending on the authentication mode, which is the same as that in the HTTPS-compliant inference service. Click **Connect** in the upper right corner to establish a WebSocket connection.

Figure 3-34 Obtaining the API address
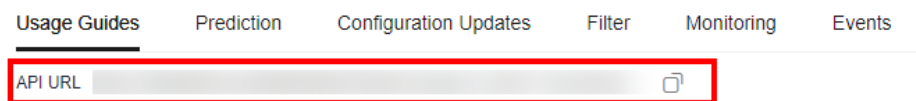
📖 **NOTE**

- – If the information is correct, **CONNECTED** will be displayed in the lower right corner.
- – If establishing the connection failed and the status code is 401, check the authentication.
- – If a keyword such as **WRONG_VERSION_NUMBER** is displayed, check whether the port configured in the custom image is the same as that configured in WebSocket or WebSocket Secure.

The following shows an established WebSocket connection.

Figure 3-35 Connection established
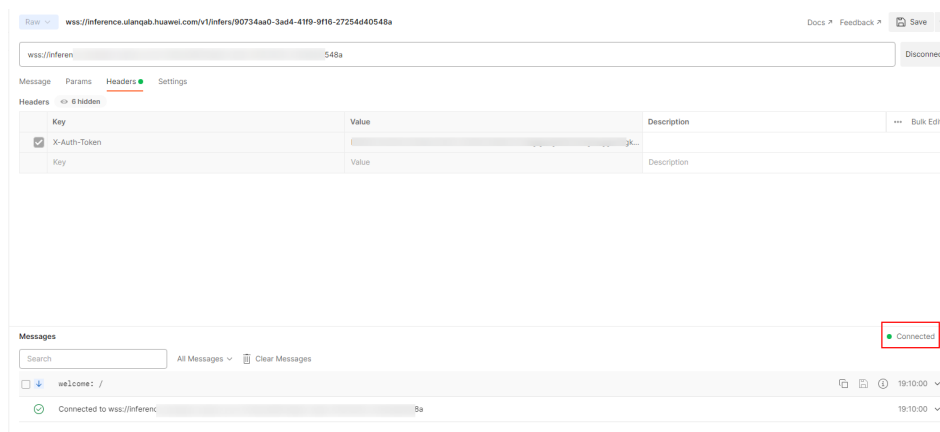
NOTICE

Preferentially check the WebSocket service provided by the custom image. The type of implementing WebSocket varies depending on the tool you used. Possible issues are as follows: A WebSocket connection can be established but cannot be maintained, or the connection is interrupted after one request and needs to be reconnected. ModelArts only ensures that it will not affect the WebSocket status in a custom image (the API address and authentication mode may be changed on ModelArts).

Step 2 Exchange data between the WebSocket client and the server.

After the connection is established, WebSocket uses TCP for full-duplex communication. The WebSocket client sends data to the server. The implementation types vary depending on the client, and the lib package may also be different for the same language. Different implementation types are not considered here.

The format of the data sent by the client is not limited by the protocol. Postman supports text, JSON, XML, HTML, and Binary data. Take text as an example. Enter the text data in the text box and click **Send** on the right to send the request to the server. If the text is oversized, Postman may be suspended.

Figure 3-36 Sending data
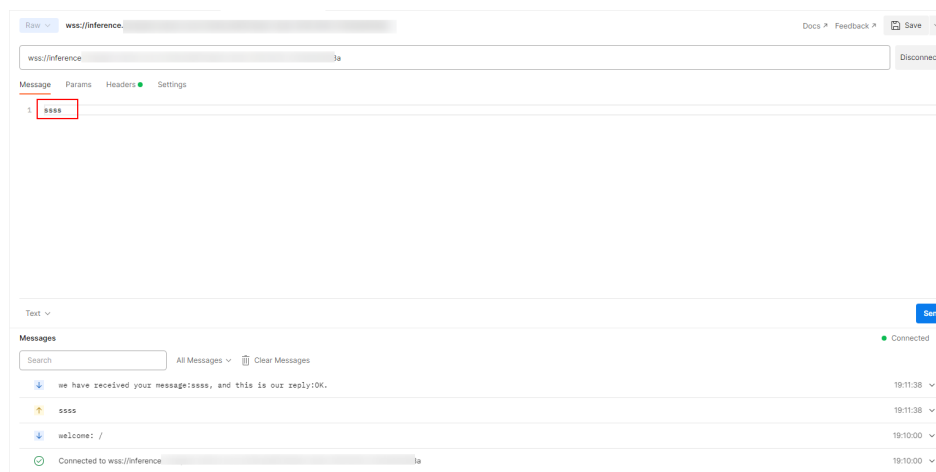
----End

3.1.4.5 Server-Sent Events

Context

Server-Sent Events (SSE) is a server push technology enabling a server to push events to a client via an HTTP connection. This technology is usually used to enable a server to push real-time data to a client, for example, a chat application or a real-time news update.

SSE primarily facilitates unidirectional real-time communication from the server to the client, such as streaming ChatGPT responses. In contrast to WebSockets, which provide bidirectional real-time communication, SSE is designed to be more lightweight and simpler to implement.

Prerequisites

The image for importing the AI application is SSE-compliant.

Constraints

- SSE supports only the deployment of real-time services.
- It supports only real-time services deployed using AI applications imported from custom images.

Calling an SSE Real-Time Service

The SSE protocol itself does not introduce new authentication mechanisms; it relies on the same methods as HTTP requests.

You can use one of the following authentication methods provided by ModelArts:

- Access Authenticated Using a Token
- Access Authenticated Using an AK/SK
- Access Authenticated Using an Application

The following section uses GUI software Postman for prediction and token authentication as an example to describe how to call an SSE service.

Figure 3-37 Calling an SSE service
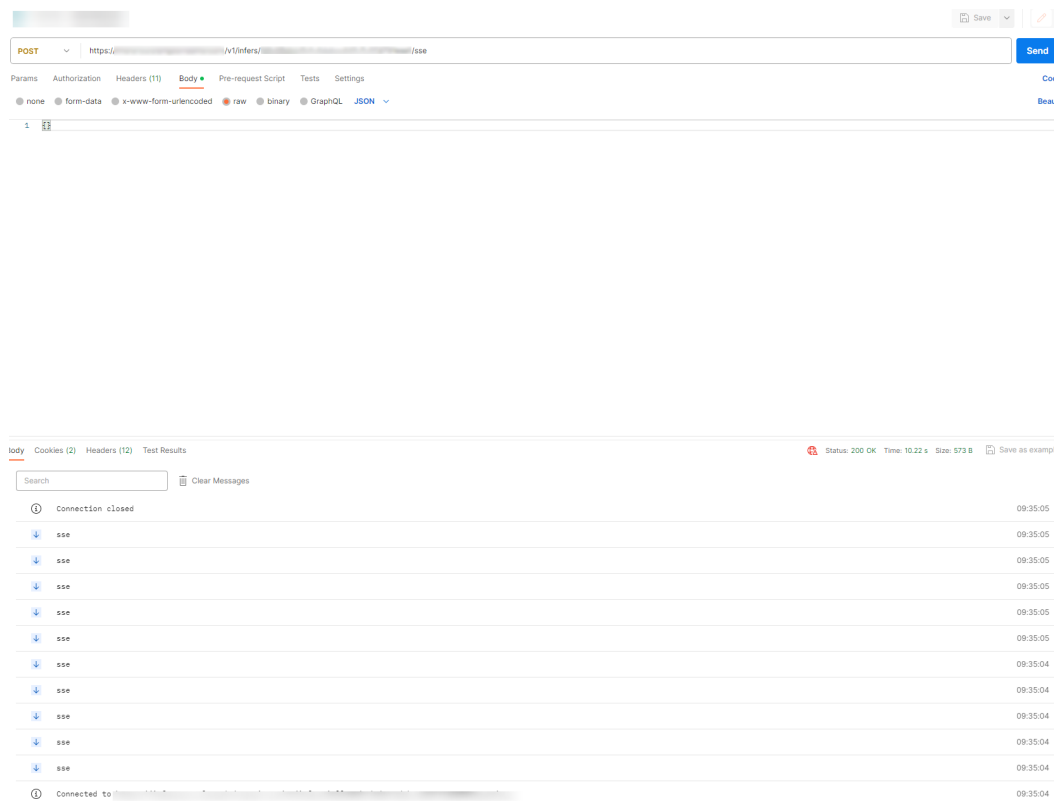
Figure 3-38 Response header **Content-Type**

📖 **NOTE**

In normal cases, the value of **Content-Type** in the response header is **text/event-stream;charset=UTF-8**.

3.1.5 Integrating a Real-Time Service

For a real-time service API that has been commissioned, you can integrate it into the production environment.

Prerequisites

The real-time service is running. Otherwise, applications in the production environment will be unavailable.

Integration Mode

ModelArts real-time services provide standard RESTful APIs, which can be accessed using HTTPS. ModelArts provides SDKs for calling real-time service APIs. For

details about how to call the SDKs, see "Scenario 1: Perform an inference test using the predictor" in *SDK Reference*.

In addition, you can use common development tools and languages to call the APIs. You can search for and obtain the guides for calling standard RESTful APIs on the Internet.

3.1.6 Cloud Shell

Scenarios

You can use Cloud Shell provided by the ModelArts console to log in to a running real-time service instance container.

Constraints

- Cloud Shell can only access a container when the associated real-time service is deployed within a dedicated resource pool
- Cloud Shell can only access a container when the associated real-time service is running.

Using Cloud Shell

1. Log in to the ModelArts console. In the navigation pane, choose **Service Deployment** > **Real-Time Services**.

2. On the real-time service list page, click the name or ID of the target service. The real-time service details page is displayed.

3. Click the **Cloud Shell** tab and select the AI application version and compute node. When the connection status changes to ✅ , you have logged in to the instance container.

   If the server disconnects due to an error or remains idle for 10 minutes, you can select **Reconnect** to regain access to the container instance.

   Figure 3-39 Cloud Shell
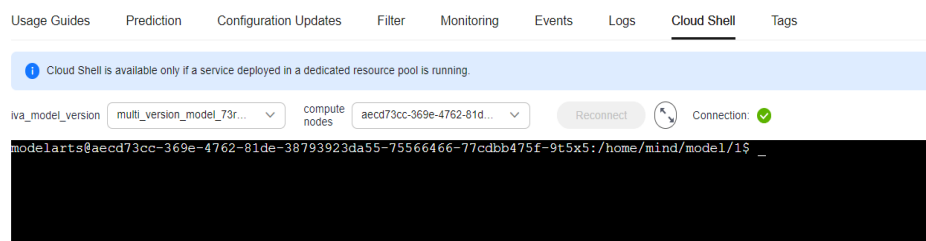
   📖 **NOTE**

   A path display exception may occur when you log in to the Cloud Shell page. In this case, press **Enter** to rectify the fault.

   Figure 3-40 Abnormal path
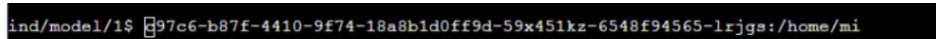
3.2 Deploying AI Applications as Batch Services

3.2.1 Deploying as a Batch Service

After an AI application is prepared, you can deploy it as a batch service. The **Service Deployment > Batch Services** page lists all batch services.

Prerequisites

- A ModelArts application in the **Normal** state is available.
- Data to be batch processed is ready and has been upload to an OBS directory.
- At least one empty folder has been created in OBS for storing the output.

Context

- A maximum of 1,000 batch services can be created.
- Based on the input request (JSON or file) defined by the AI application, different parameters are entered. If the AI application input is a JSON file, a configuration file is required to generate a mapping file. If the AI application input is a file, no mapping file is required.
- Batch services can only be deployed in a public resource pool, but not a dedicated resource pool.

Procedure

1. Log in to the ModelArts management console. In the left navigation pane, choose **Service Deployment** > **Batch Services**. By default, the **Batch Services** page is displayed.

2. In the batch service list, click **Deploy** in the upper left corner. The **Deploy** page is displayed.

3. Set parameters for a batch service.

   a. Set the basic information, including **Name** and **Description**. The name is generated by default, for example, **service-bc0d**. You can specify **Name** and **Description** according to actual requirements.

   b. Set other parameters, including the resource pool and AI application configurations. For details, see **Table 3-11**.

   Table 3-11 Parameters

Parameter	Description
AI Application Source	Select **My AI Applications** or **My Subscriptions** based on your requirements.
AI Application and Version	Select an AI application and version that are running properly.

Parameter	Description
Input Path	Select the OBS directory where the uploaded data is stored. Select a folder or a .manifest file. For details about the specifications of the .manifest file, see **Manifest File Specifications**. **NOTE** ● If the input data is an image, ensure that the size of a single image is less than 12 MB. ● If the input data is in CSV format, ensure that no Chinese character is included. ● If the input data is in CSV format, ensure that the file size does not exceed 12 MB. ● If an image or CSV file is larger than 12 MB, an error is reported. In this case, resize the file or contact technical support to adjust the file size limit.
Request Path	URL used for calling the AI application API in a batch service, and also the request path of the AI application service. Its value is obtained from the **url** field of **apis** in the AI application configuration file.
Mapping Relationship	If the AI application input is in JSON format, the system automatically generates the mapping based on the configuration file corresponding to the AI application. If the AI application input is other file, mapping is not required. Automatically generated mapping file. Enter the field index corresponding to each parameter in the CSV file. The index starts from 0. Mapping rule: The mapping rule comes from the input parameter (**request**) in the model configuration file **config.json**. When **type** is set to **string, number, integer, or boolean**, you are required to set the index parameter. For details about the mapping rule, see **Example Mapping**. The index must be a positive integer starting from 0. If the value of index does not comply with the rule, this parameter is ignored in the request. After the mapping rule is configured, the corresponding CSV data must be separated by commas (,).
Output Path	Select the path for saving the batch prediction result. You can select the empty folder that you create.

Parameter	Description
Specifications	The system provides available compute resources matching your AI application. Select an available resource from the drop-down list. For example, if the model comes from an ExeML project, the compute resources are automatically associated with the ExeML specifications for use.
Compute Nodes	Set the number of instances for the current AI application version. If you set the number of nodes to **1**, the standalone computing mode is used. If you set the number of nodes to a value greater than 1, the distributed computing mode is used. Select a computing mode based on the actual requirements.
Environment Variable	Set environment variables and inject them to the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables.
Timeout	Timeout of a single model, including both the deployment and startup time. The default value is 20 minutes. The value must range from 3 to 120.
Runtime Log Output	This function is disabled by default. The run logs of batch services are stored only in the ModelArts log system. You can query the run logs on the **Logs** tab page of the service details page. If this function is enabled, the run logs of batch services are exported and stored in Log Tank Service (LTS). LTS automatically creates log groups and log streams and caches run logs generated within seven days by default. For details about the LTS log management function, see **Log Tank Service**. **NOTE** ● This cannot be disabled once it is enabled. ● You will be billed for the log query and log storage functions provided by LTS. For details, see section **LTS Pricing Details**. ● Do not print unnecessary audio log files. Otherwise, system logs may fail to be displayed, and the error message "Failed to load audio" may be displayed.

4. After setting the parameters, deploy the model as a batch service as prompted. Deploying a service generally requires a period of time, which may be several minutes or tens of minutes depending on the amount of your data and resources.

📖 **NOTE**

After a batch service is deployed, it is started immediately. During the running, you will be charged based on your selected resources.

You can go to the batch service list to view the basic information about the batch service. In the batch service list, after the status of the newly deployed service changes from **Deploying** to **Running**, the service is deployed successfully.

Manifest File Specifications

ModelArts batch services support manifest files, which describe data input and output.

Example input manifest file

- File name: **test.manifest**

- File content:
```
{"source": "obs://test/data/1.jpg"}
{"source": "s3://test/data/2.jpg"}
{"source": "https://infers-data.obs.cn-north-1.myhuaweicloud.com:443/xgboosterdata/data.csv?
AccessKeyId=2Q0V0TQ461N26DDL18RB&Expires=1550611914&Signature=wZBttZj5QZrReDhz1uDzwve
8GpY%3D&x-obs-security-token=gQpzb3V0aGNoaW5hixvY8V9a1SnsxmGoHYmB1SArYMyqnQT-
ZaMSxHvl68kKLAy5feYvLDM..."}
```

- Requirements on the file:

  a. The file name extension must be **.manifest**.

  b. The file content is in JSON format. Each row describes a piece of input data, which must be accurate to a file instead of a folder.

  c. A **source** field must be defined for the JSON content. The field value is the OBS URL of the file in any of the following formats:

     i. Bucket path *<obs path>{{Bucket name}}/{{Object name}}/File name*, which is used to access your OBS data. You can access the path for obtaining an object in OBS. *<obs path>* can be **obs://** or **s3://**.

     ii. Shared link generated by OBS, including signature information. It applies to accessing OBS data of other users. The link has a validity period. Perform operations within the period.

Example output manifest file

A manifest file will be generated in the output directory of the batch services.

- Assume that the output path is **//test-bucket/test/**. The result is stored in the following path:
```
OBS bucket/directory name
├── test-bucket
│   ├── test
│   │   ├── infer-result-{{task_id}}.manifest
│   │   ├── infer-result
│   │   │   ├── 1.jpg_result.txt
│   │   │   ├── 2.jpg_result.txt
```
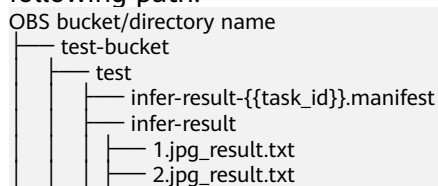
- Content of the **infer-result-0.manifest** file:
```
{"source": "obs://obs-data-bucket/test/data/1.jpg","result":"SUCCESSFUL","inference-loc": "obs://test-
bucket/test/infer-result/1.jpg_result.txt"}
{"source": "s3://obs-data-bucket/test/data/2.jpg","result":"FAILED","error_message": "Download file
failed."}
{"source ": "https://infers-data.obs.xxx.com:443/xgboosterdata/2.jpg?
AccessKeyId=2Q0V0TQ461N26DDL18RB&Expires=1550611914&Signature=wZBttZj5QZrReDhz1uDzwve
8GpY%3D&x-obs-security-token=gQpzb3V0aGNoaW5hixvY8V9a1SnsxmGoHYmB1SArYMyqnQT-
ZaMSxHvl68kKLAy5feYvLDMNZWxzhBZ6Q-3HcoZMh9gISwQOVBwm4ZytB_m8sg1fL6isU7T3CnoL9jmv
DGgT9VBC7dC1EyfSJrUcqfB_N0ykCsfrA1Tt_IQYZFDu_HyqVk-
GunUcTVdDfWlCV3TrYcpmznZjliAnYUO89kAwCYGeRZsCsC0ePu4PHMsBvYV9gWmN9AUZIDn1sfRL4vo
BpwQnp6tnAgHW49y5a6hP2hCAoQ-95SpUriJ434QlymoeKfTHVMKOeZxZea-
JxOvevOCGI5CcGehEJaz48sgH81UiHzl21zocNB_hpPfus2jY6KPglEJxMv6Kwmro-
```

ZBXWuSJUDOnSYXI-3ciYjg9-
h10b8W3sW1mOTFCWNGoWsd74it7l_5-7UUhoIeyPByO_REwkur2FOJsuMpGlRaPyglZxXm_jfdLFXobYtz
Zhbul4yWXga6oxTOkfcwykTOYH0NPoPRt5MYGYweOXXxFs3d5w2rd0y7p0QYhyTzIkk5CIz7FlWNapFISL
7zdhsl8RfchTqESq94KgkeqatSF_iIvnYMW2r8P8x2k_eb6NJ7U_q5ztMbO9oWEcfr0D2f7n7Bl_nb2HIB_H9tj
zKvqwngaimYhBbMRPfibvttW86GiwVP8vrC27FOn39Be9z2hSfJ_8pHej0yMlyNqZ481FQ5vWT_vFV3JHM-
7I1ZB0_hIdaHfltm-J69cTfHSEOzt7DGaMIES1o7U3w%3D%3D","result":"SUCCESSFUL","inference-loc":
"obs://test-bucket/test/infer-result/2.jpg_result.txt"}

- File format:

  a. The file name is **infer-result-{{task_id}}.manifest**, where **task_id** is the batch task ID, which is unique for a batch service.

  b. If a large number of files need to be processed, multiple manifest files may be generated with the same suffix **.manifest** and are distinguished by suffix, for example, **infer-result-{{task_id}}_1.manifest**.

  c. The **infer-result**-*{{task_id}}* directory is created in the manifest directory to store the file processing result.

  d. The file content is in JSON format. Each row describes the output result of a piece of input data.

  e. The file contains multiple fields:

     i. **source**: input data description, which is the same as that of the input manifest file

     ii. **result**: file processing result, which can be **SUCCESSFUL** or **FAILED**

     iii. inference-loc: output result path. This field is available when result is **SUCCESSFUL**. The format is **obs://{{**Bucket name**}}/{**Object name**}**.

     iv. error_message: error information. This field is available when the result is FAILED.

Example Mapping

The following example shows the relationship between the configuration file, mapping rule, CSV data, and inference request.

The following uses a file for prediction as an example:

```
[
  {
    "method": "post",
    "url": "/",
    "request": {
      "Content-type": "multipart/form-data",
      "data": {
        "type": "object",
        "properties": {
          "data": {
            "type": "object",
            "properties": {
              "req_data": {
                "type": "array",
                "items": [
                  {
                    "type": "object",
                    "properties": {
                      "input_1": {
                        "type": "number"
                      },
                      "input_2": {
                        "type": "number"
                      },
                      "input_3": {
```

```
                "type": "number"
            },
            "input_4": {
                "type": "number"
            }
        }
      }
    ]
  }
}
}
}
}
}
}
}
}
]
```

The ModelArts management console automatically resolves the mapping relationship from the configuration file as shown below. When calling a ModelArts API, configure the mapping by following the rule.

```
{
    "type": "object",
    "properties": {
        "data": {
            "type": "object",
            "properties": {
                "req_data": {
                    "type": "array",
                    "items": [
                        {
                            "type": "object",
                            "properties": {
                                "input_1": {
                                    "type": "number",
                                    "index": 0
                                },
                                "input_2": {
                                    "type": "number",
                                    "index": 1
                                },
                                "input_3": {
                                    "type": "number",
                                    "index": 2
                                },
                                "input_4": {
                                    "type": "number",
                                    "index": 3
                                }
                            }
                        }
                    ]
                }
            }
        }
    }
}
```

Multiple pieces of CSV data for inference are separated by commas (,) The following shows an example:

```
5.1,3.5,1.4,0.2
4.9,3.0,1.4,0.2
4.7,3.2,1.3,0.2
```

Depending on the defined mapping relationship, the inference request is shown below, whose format is similar to that for real-time services.

```
{
    "data": {
```

```
        "req_data": [{
            "input_1": 5.1,
            "input_2": 3.5,
            "input_3": 1.4,
            "input_4": 0.2
        }]
    }
}
```

3.2.2 Viewing Details About a Batch Service

After an AI application is deployed as a batch service, you can access the **Batch Services** page to view its details.

1. Log in to the ModelArts console and choose **Service Deployment** > **Batch Services**.
2. Click the name of the target service. The service details page is displayed.

   View service information. For details, see **Table 3-12**.

Table 3-12 Batch service parameters

Parameter	Description
Name	Name of the batch service.
Service ID	ID of the batch service.
Status	Status of the batch service.
Job ID	Job ID of the batch service.
Instance Flavor	Node flavor of the batch service.
Compute Nodes	Number of nodes of the batch service.
Start Time	Start time of the batch service job.
Environment Variable	Environment variables added during batch service creation.
End Time	End time of the batch service job.
Description	Service description, which you can click the edit button to modify.
Input Path	OBS path to the input data in the batch service.
Output Path	OBS path to the output data in the batch service.
AI Application Name & Version	Name and version of the AI application used by the batch service.

Parameter	Description
Advanced Log Management	This function is disabled by default. The runtime logs of batch services are stored only in the ModelArts log system. If this function is enabled, the runtime logs of batch services will be exported and stored in Log Tank Service (LTS). LTS automatically creates log groups and log streams and caches run logs generated within seven days by default. For details about the LTS log management function, see **Log Tank Service**. **NOTE** ● This cannot be disabled once it is enabled. ● You will be billed for the log query and log storage functions provided by LTS. For details, see section **LTS Pricing Details**. ● Do not print unnecessary audio log files. Otherwise, system logs may fail to be displayed, and the error message "Failed to load audio" may be displayed.

3. Switch between tabs on the details page of a batch service to view more details. For details, see **Table 3-13**.

Table 3-13 Batch service tabs

Parameter	Description
Events	This page displays key operations during service use, such as the service deployment progress, detailed causes of deployment exceptions, and time points when a service is started, stopped, or modified. Events are saved for one month and will be automatically cleared then. For details about how to view events of a service, see **Viewing Service Events**.

Parameter	Description
Logs	This page displays the log information about each AI application in the service. You can view logs generated in the latest 5 minutes, latest 30 minutes, latest 1 hour, and user-defined time segment.
	You can select the start time and end time when defining a time segment.
	If this function is enabled, the logs stored in LTS will be displayed. You can click **View Complete Logs on LTS** to view all logs.
	Log search rules:
	● Do not enter a string that contains any of the following delimiters: ,'";=()[]{}@&<>/:\n\t\r.
	● You can use exact search by keyword. A keyword refers to the word between two adjacent delimiters.
	● You can use fuzzy search by keyword. For example, you can enter **error**, **er?or**, **rro***, or **er*r**.
	● You can enter a phrase for exact search. For example, **Start to refresh**.
	● Before enabling this function, you can combine keywords with **&&** or **||**. For example, **query logs&&erro*** or **query logs||erro***. After enabling this function, you can combine keywords with **AND** or **OR**. For example, **query logs AND erro*** or **query logs OR erro***.

3.2.3 Viewing the Batch Service Prediction Result

When deploying a batch service, you can select the location of the output data directory. You can view the running result of the batch service that is in the **Completed** status.

Procedure

1. Log in to the ModelArts management console and choose **Service Deployment** > **Batch Services**.
2. Click the name of the target service in the **Completed** status. The service details page is displayed.
   - You can view the service name, status, ID, input path, output path, and description.
   - You can click in the **Description** area to edit the description.
3. Obtain the detailed OBS path next to **Output Path**, switch to the path and obtain the batch service prediction results, including the prediction result file and the AI application prediction result.

If the prediction is successful, the directory contains the prediction result file and AI application prediction result. Otherwise, the directory contains only the prediction result file.

- Prediction result file: The file is in *xxx*.**manifest** format, which contains the file path and prediction result, and more.

- AI application prediction result:

  - If images are input, a result file is generated for each image in the *Image name*__result.txt format, for example, **IMG_20180919_115016.jpg_result.txt**.

  - If audio files are input, a result file is generated for each audio file in the *Audio file name*__**result.txt** format, for example, **1-36929-A-47.wav_result.txt**.

  - If table data is input, the result file is generated in the *Table name*__**result.txt** format, for example, **train.csv_result.txt**.

3.3 Upgrading a Service

For a deployed service, you can modify its basic information to match service changes and change the AI application version to upgrade it.

You can modify the basic information about a service in either of the following ways:

Method 1: Modify Service Information on the Service Management Page

Method 2: Modify Service Information on the Service Details Page

Prerequisites

The service has been deployed. The service in the **Deploying** state cannot be upgraded by modifying the service information.

Constraints

- Improper upgrade operations will interrupt service running during the upgrade. Therefore, exercise caution when performing this operation.

- ModelArts supports hitless rolling upgrade of real-time services in some scenarios. Before upgrade, prepare for it and confirm the prerequisites.

Table 3-14 Scenarios for hitless rolling upgrade

Meta Model Source for Creating an AI Application	Using a Public Resource Pool	Using a Dedicated Resource Pool
Training job	Not supported	Not supported
Template	Not supported	Not supported

Meta Model Source for Creating an AI Application	Using a Public Resource Pool	Using a Dedicated Resource Pool
Container image	Not supported	Supported. The custom image for creating an AI application must meet **Custom Image Specifications for Creating AI Applications**.
OBS	Not supported	Not supported

Method 1: Modify Service Information on the Service Management Page

1. Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service.

2. In the service list, click **Modify** in the **Operation** column of the target service, modify basic service information, and submit the modification task as prompted.

   When some parameters are modified, the system automatically restarts the service for the modification to take effect. When you submit a service modification task, if a restart is required, a dialog box will be displayed.

   – For details about the real-time service parameters, see **Deploying as a Real-Time Service**. To modify a real-time service, you also need to set **Max. Invalid Instances** to set the maximum number of nodes that can be concurrently upgraded, during which time these nodes are invalid.

   – For details about the batch service parameters, see **Deploying as a Batch Service**.

Method 2: Modify Service Information on the Service Details Page

1. Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service.

2. Click the name of the target service. The service details page is displayed.

3. Click **Modify** in the upper right corner of the page, modify the service details, and submit the modification task as prompted.

   When some parameters are modified, the system automatically restarts the service for the modification to take effect. When you submit a service modification task, if a restart is required, a dialog box will be displayed.

   – For details about the real-time service parameters, see **Deploying as a Real-Time Service**. To modify a real-time service, you also need to set **Max. Invalid Instances** to set the maximum number of nodes that can be concurrently upgraded, during which time these nodes are invalid.

   – For details about the batch service parameters, see **Deploying as a Batch Service**.

3.4 Starting, Stopping, Deleting, or Restarting a Service

Starting a Service

You can start services in the **Successful**, **Abnormal**, or **Stopped** status. Services in the **Deploying** state cannot be started. A service is billed when it is started and in the **Running** state. You can start a service in the following ways:

- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click **Start** in the **Operation** column to start the target service.

- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click the name of the target service. The service details page is displayed. Click **Start** in the upper right corner of the page to start the service.

📖 **NOTE**

Services deployed on ModelArts edge nodes or ModelArts edge resource pools cannot be started.

Stopping a Service

A stopped service will no longer be billed. Stop a service in either of the following ways:

- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click **Stop** in the **Operation** column to stop a service. (For a real-time service, choose **More** > **Stop** in the **Operation** column.)

- Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service. Click the name of the target service. The service details page is displayed. Click **Stop** in the upper right corner of the page to stop the service.

📖 **NOTE**

Services deployed on ModelArts edge nodes or ModelArts edge resource pools cannot be stopped.

Deleting a Service

If a service is no longer in use, delete it to release resources.

Log in to the ModelArts management console and choose **Service Deployment** from the left navigation pane. Go to the service management page of the target service.

- Real-time services

- In the real-time service list, choose **More** > **Delete** in the **Operation** column of the target service to delete it.

- Select services in the real-time service list and click **Delete** above the list to delete services in batches.

- Click the name of the target service. On the displayed service details page, click **Delete** in the upper right corner to delete the service.

- Batch services

  - In the batch service list, click **Delete** in the **Operation** column of the target service to delete it.

  - Select services in the batch service list and click **Delete** above the list to delete services in batches.

  - Click the name of the target service. On the displayed service details page, click **Delete** in the upper right corner to delete the service.

◯ NOTE

- A deleted service cannot be recovered.

- A service cannot be deleted without agency authorization.

- If **Advanced Log Management** is enabled for a real-time service, you are advised to delete the LTS logs and streams when you delete the service. This prevents additional fees incurred by the logs and streams.

Restarting a Service

You can restart a real-time service only when the service is in the **Running** or **Alarm** state. Batch services and edge services cannot be restarted. You can restart a real-time service in either of the following ways:

- Log in to the ModelArts management console and choose **Service Deployment** from the navigation pane. Go to the real-time service list page. Click **More** > **Restart** in the **Operation** column to restart the target service.

- Log in to the ModelArts management console and choose **Service Deployment** from the navigation pane. Go to the real-time service list page. Click the name of the target service. The service details page is displayed. Click **Restart** in the upper right corner of the page to restart the service.

3.5 Viewing Service Events

During the whole lifecycle of a service, every key event is automatically recorded. You can view the events on the details page of the service at any time.

This helps you better understand the process of deploying a service and locate faults more accurately when a task exception occurs. The following table lists the available events.

Table 3-15 Events

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	The service starts to deploy.	-

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	Insufficient resources. Wait until idle resources are sufficient.	Wait until the resources are released and try again.
Abnormal	Insufficient *xxx*. The scheduling failed. Supplementary information: *xxx*	Learn about resource insufficiency details based on the supplementary information. For details, see **FAQs**.
Normal	The image starts to create.	-
Abnormal	Failed to create model image **xxx**. For details, see logs *:|nxxx*.	Locate and rectify the fault based on the build logs.
Abnormal	Failed to create the image.	Contact technical support.
Normal	The image created.	-
Abnormal	Service *xxx* failed. Error: *xxx*	Locate and rectify the fault based on the error information.
Abnormal	Failed to update the service. Perform a rollback.	Contact technical support.
Normal	The service is being updated.	-
Normal	The service is being started.	-
Normal	The service is being stopped.	-
Normal	The service has been stopped.	-
Normal	Auto stop has been disabled.	-
Normal	Auto stop has been enabled. The service will stop after *x*s.	-
Normal	The service stops when the auto stop time expires.	-

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	The service is stopped because the quota exceeds the upper limit.	Contact technical support.
Abnormal	Failed to automatically stop the service. Error: *xxx*	Locate and rectify the fault based on the error information.
Normal	Service instances deleted from resource pool *xxx*.	-
Normal	Service instances stopped in resource pool *xxx*.	-
Abnormal	The batch service failed. Try again later. Error: *xxx*	Locate and rectify the fault based on the error information.
Normal	The service has been executed.	-
Abnormal	Failed to stop the service. Error: *xxx*	Locate and rectify the fault based on the error information.
Normal	The subscription license *xxx* is to expire.	-
Normal	Service *xxx* started.	-
Abnormal	Failed to start service *xxx*.	For details about how to locate and rectify the fault, see **FAQs**.
Abnormal	Service deployment timed out. Error: *xxx*	Locate and rectify the fault based on the error information.
Normal	Failed to update the service. The update has been rolled back.	-
Abnormal	Failed to update the service. The rollback failed.	Contact technical support.

During service deployment and running, key events can both be manually and automatically refreshed.

Viewing Events

1. In the left navigation pane of the ModelArts management console, choose **Service Deployment** > **Real-Time Services** or **Batch Services** or **Edge Services**. In the service list, click the name or ID of the target service to go to its details page.

2. View the events on the **Events** tab page.

4 Inference Specifications

Model Package Specifications

Model Templates

Examples of Custom Scripts

4.1 Model Package Specifications

4.1.1 Introduction to Model Package Specifications

When creating an AI application, make sure that any meta model imported from OBS complies with specifications.

> **NOTE**
>
> - The model package specifications are used when you import one model. If you import multiple models, for example, there are multiple model files, use custom images.
> - If you want to use an AI engine that is not supported by ModelArts, use a custom image.
> - For details about how to create a custom image, see **Custom Image Specifications for Creating AI Applications** and **Creating a Custom Image and Using It to Create an AI Application**.
> - For more examples of custom scripts, see **Examples of Custom Scripts**.

The model package must contain the **model** directory. The **model** directory stores the model file, model configuration file, and model inference code file.

- **Model files:** The requirements for model files vary according to the model package structure. For details, see **Model Package Example**.

- **Model configuration file**: The model configuration file must be available and its name is consistently to be **config.json**. There must be only one model configuration file. For details about how to edit a model configuration file, see **Specifications for Editing a Model Configuration File** .

- **Model inference code file**: It is mandatory. The file name is consistently to be **customize_service.py**. There must be only one model inference code file. For details about how to edit model inference code, see **Specifications for Writing Model Inference Code** .

– The **.py** file on which **customize_service.py** depends can be directly stored in the **model** directory. Use a relative import mode to import the custom package.

– The other files on which **customize_service.py** depends can be stored in the **model** directory. You must use absolute paths to access these files. For more details, see **Obtaining an Absolute Path**.

ModelArts provides samples and sample code for multiple engines. You can compile your configuration files and inference code by referring to **ModelArts Samples**. ModelArts also provides custom script examples of common AI engines. For details, see **Examples of Custom Scripts**.

If you encounter any problem when importing a meta model, **contact Huawei Cloud technical support**.

Model Package Example

- Structure of the TensorFlow-based model package

  When publishing the model, you only need to specify the **ocr** directory.

  ```
  OBS bucket/directory name
  ├── ocr
  │   ├── model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
  │   │   ├── <<Custom Python package>> (Optional) User's Python package, which can be directly referenced in model inference code
  │   │   ├── saved_model.pb (Mandatory) Protocol buffer file, which contains the diagram description of the model
  │   │   ├── variables Name of a fixed sub-directory, which contains the weight and deviation rate of the model. It is mandatory for the main file of the *.pb model.
  │   │   │   ├── variables.index Mandatory
  │   │   │   ├── variables.data-00000-of-00001 Mandatory
  │   │   ├──config.json (Mandatory) Model configuration file. The file name is fixed to config.json. Only one model configuration file is supported.
  │   │   ├──customize_service.py (Mandatory) Model inference code. The file name is fixed to customize_service.py. Only one model inference code file exists.
  The files on which customize_service.py depends can be directly stored in the model directory.
  ```

- Structure of the PyTorch-based model package

  When publishing the model, you only need to specify the **resnet** directory.

  ```
  OBS bucket/directory name
  ├── resnet
  │   ├── model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
  │   │   ├── <<Custom Python package>> (Optional) User's Python package, which can be directly referenced in model inference code
  │   │   ├── resnet50.pth (Mandatory) PyTorch model file, which contains variable and weight information and is saved as state_dict
  │   │   ├──config.json (Mandatory) Model configuration file. The file name is fixed to config.json. Only one model configuration file is supported.
  │   │   ├──customize_service.py (Mandatory) Model inference code. The file name is fixed to customize_service.py. Only one model inference code file exists. The files on which customize_service.py depends can be directly stored in the model directory.
  ```

- Structure of a custom model package depends on the AI engine in your custom image. For example, if the AI engine in your custom image is TensorFlow, the model package uses the TensorFlow structure.

4.1.2 Specifications for Editing a Model Configuration File

A model developer needs to edit a configuration file **config.json** when publishing a model. The model configuration file describes the model usage, computing framework, precision, inference code dependency package, and model API.

Configuration File Format

The configuration file is in JSON format. **Table 4-1** describes the parameters.

Table 4-1 Parameters

Parameter	Mandatory	Data Type	Description
model_algorithm	Yes	String	Model algorithm, which is set by the model developer to help model users understand the usage of the model. The value must start with a letter and contain no more than 36 characters. Chinese characters and special characters (&!'\"<>=) are not allowed. Common model algorithms include **image_classification** (image classification), **object_detection** (object detection), and **predict_analysis** (prediction analysis).
model_type	Yes	String	Model AI engine, which indicates the computing framework used by a model. Common AI engines and **Image** are supported. • For details about supported AI engines, see **Supported AI Engines for ModelArts Inference**. • If **model_type** is set to **Image**, the AI application is created using a custom image. In this case, parameter **swr_location** is mandatory. For details about specifications for custom images, see **Custom Image Specifications for Creating AI Applications**.
runtime	No	String	Model runtime environment. Python2.7 is used by default The value of **runtime** depends on the value of **model_type**. If **model_type** is set to **Image**, you do not need to set **runtime**. If **model_type** is set to another mainstream framework, select the engine and runtime environment. For details about the supported running environments, see **Supported AI Engines for ModelArts Inference**. If your model must run on specified CPUs or GPUs, select the CPUs or GPUs based on the runtime suffix. If the runtime does not contain the CPU or GPU information, check the runtime description in *Supported AI Engines for ModelArts Inference*.

Paramete r	Mand atory	Data Type	Description
metrics	No	Objec t	Model precision information, including the average value, recall rate, precision, and accuracy. For details about the **metrics** object structure, see **Table 4-2**. The result is displayed in the model precision area on the AI application details page.
apis	No	api array	Format of the requests received and returned by a model. The value is structure data. It is the RESTful API array provided by a model. For details about the API data structure, see **Table 4-3**. For details about the code example, see **Code Example of apis Parameters**. ● If **model_type** is set to Image, the AI application is created using a custom image. ● When **model_type** is not **Image**, only one API whose request path is **/** can be declared in **apis** because the preconfigured AI engine exposes only one inference API whose request path is **/**.
dependen cies	No	depen dency array	Package on which the model inference code depends, which is structure data. Model developers need to provide the package name, installation mode, and version constraints. Only the pip installation mode is supported. **Table 4-6** describes the dependency array. If the model package does not contain the **customize_service.py** file, you do not need to set this parameter. Dependency packages cannot be installed for custom image models. **NOTE** The **dependencies** parameter supports multiple dependency structure arrays in list format and applies to scenarios where the default installation packages have dependency relationships. Packages on the top are installed first. The wheel package on premises can be used for installation. (The wheel package must be stored in the same directory as the model file). For details, see **How Do I Edit the Installation Package Dependency Parameters in a Model Configuration File When Importing a Model?**

Paramete r	Mand atory	Data Type	Description
health	No	**health** data struct ure	Configuration of an image health interface. This parameter is mandatory only when **model_type** is set to **Image**. If services cannot be interrupted during a rolling upgrade, a health check API must be provided for ModelArts to call. For details about the health data structure, see **Table 4-8**.

Table 4-2 metrics object description

Paramete r	Mand atory	Data Type	Description
f1	No	Numb er	F1 score. The value is rounded to 17 decimal places.
recall	No	Numb er	Recall rate. The value is rounded to 17 decimal places.
precision	No	Numb er	Precision. The value is rounded to 17 decimal places.
accuracy	No	Numb er	Accuracy. The value is rounded to 17 decimal places.

Table 4-3 api array

Paramet er	Manda tory	Data Type	Description
url	No	String	Request path. The default value is a slash (**/**). For a custom image model (**model_type** is **Image**), set this parameter to the actual request path exposed in the image. For a non-custom image model (**model_type** is not **Image**), the URL can only be **/**.
method	No	String	Request method. The default value is **POST**.
request	No	Object	Request body. For details, see **Table 4-4**.
response	No	Object	Response body. For details, see **Table 4-5**.

Table 4-4 request description

Paramet er	Mandat ory	Data Type	Description
Content-type	No for real-time services Yes for batch services	String	Data is sent in a specified content format. The default value is **application/json**. The options are as follows: • **application/json**: JSON data is uploaded. • **multipart/form-data**: A file is uploaded. **NOTE** For machine learning models, only **application/json** is supported.
data	No for real-time services Yes for batch services	String	The request body is described in JSON schema. For details about the parameter description, see the **official guide**.

Table 4-5 response description

Paramet er	Mandat ory	Data Type	Description
Content-type	No for real-time services Yes for batch services	String	Data is sent in a specified content format. The default value is **application/json**. **NOTE** For machine learning models, only **application/json** is supported.
data	No for real-time services Yes for batch services	String	The response body is described in JSON schema. For details about the parameter description, see the **official guide**.

Table 4-6 dependency array

Parameter	Mandatory	Data Type	Description
installer	Yes	String	Installation method. Only **pip** is supported.

Parameter	Mandatory	Data Type	Description
packages	Yes	package array	Dependency package collection. For details about the package structure array, see **Table 4-7**.

Table 4-7 package array

Parameter	Mandatory	Type	Description
package_na me	Yes	String	Dependency package name. Chinese characters and special characters (&!'"<>=) are not allowed.
package_ver sion	No	String	Dependency package version. If the dependency package does not rely on package versions, leave this field blank. Chinese characters and special characters (&!'"<>=) are not allowed.
restraint	No	String	Version restriction. This parameter is mandatory only when **package_version** is configured. Possible values are **EXACT**, **ATLEAST**, and **ATMOST**. ● **EXACT** indicates that a specified version is installed. ● **ATLEAST** indicates that the version of the installation package is not earlier than the specified version. ● **ATMOST** indicates that the version of the installation package is not later than the specified version. **NOTE** ● If there are specific requirements on the version, preferentially use **EXACT**. If **EXACT** conflicts with the system installation packages, you can select **ATLEAST**. ● If there is no specific requirement on the version, retain only the **package_name** parameter and leave **restraint** and **package_version** blank.

Table 4-8 health data structure description

Parameter	Mandatory	Type	Description
check_method	Yes	String	Health check method. The value can be **HTTP** or **EXEC**. ● **HTTP**: Use an HTTP request. ● **EXEC**: Execute a command.
command	No	String	Health check command. This parameter is mandatory when **check_method** is set to **EXEC**.
url	No	String	Request URL of a health check API. This parameter is mandatory when **check_method** is set to **HTTP**.
protocol	No	String	Request protocol of a health check API. The default value is **http**. This parameter is mandatory when **check_method** is set to **HTTP**.
initial_delay_seconds	No	String	Delay for initializing the health check.
timeout_seconds	No	String	Health check timeout.
period_seconds	Yes	String	Health check period, in seconds. Enter an integer greater than 0 and no more than 2147483647.
failure_threshold	Yes	String	Maximum number of health check failures. Enter an integer greater than 0 and no more than 2147483647.

Code Example of apis Parameters

```
[{
    "url": "/",
    "method": "post",
    "request": {
        "Content-type": "multipart/form-data",
        "data": {
            "type": "object",
            "properties": {
                "images": {
                    "type": "file"
                }
            }
        }
    },
    "response": {
        "Content-type": "applicaton/json",
        "data": {
```

```
                    "type": "object",
                    "properties": {
                        "mnist_result": {
                            "type": "array",
                            "item": [
                                {
                                    "type": "string"
                                }
                            ]
                        }
                    }
                }
            }
        }
    }
}]
```

Example of the Object Detection Model Configuration File

The following code uses the TensorFlow engine as an example. You can modify the **model_type** parameter based on the actual engine type.

- Model input

  Key: **images**

  Value: **image files**

- Model output

```
{
    "detection_classes": [
        "face",
        "arm"
    ],
    "detection_boxes": [
        [
            33.6,
            42.6,
            104.5,
            203.4
        ],
        [
            103.1,
            92.8,
            765.6,
            945.7
        ]
    ],
    "detection_scores": [0.99, 0.73]
}
```

- Configuration file

```
{
    "model_type": "TensorFlow",
    "model_algorithm": "object_detection",
    "metrics": {
        "f1": 0.345294,
        "accuracy": 0.462963,
        "precision": 0.338977,
        "recall": 0.351852
    },
    "apis": [{
        "url": "/",
        "method": "post",
        "request": {
            "Content-type": "multipart/form-data",
            "data": {
                "type": "object",
                "properties": {
                    "images": {
                        "type": "file"
```

```
                }
              }
            }
          },
          "response": {
            "Content-type": "application/json",
            "data": {
              "type": "object",
              "properties": {
                "detection_classes": {
                  "type": "array",
                  "items": [{
                    "type": "string"
                  }]
                },
                "detection_boxes": {
                  "type": "array",
                  "items": [{
                    "type": "array",
                    "minItems": 4,
                    "maxItems": 4,
                    "items": [{
                      "type": "number"
                    }]
                  }]
                },
                "detection_scores": {
                  "type": "array",
                  "items": [{
                    "type": "number"
                  }]
                }
              }
            }
          }
        }
      }],
      "dependencies": [{
        "installer": "pip",
        "packages": [{
            "restraint": "EXACT",
            "package_version": "1.15.0",
            "package_name": "numpy"
          },
          {
            "restraint": "EXACT",
            "package_version": "5.2.0",
            "package_name": "Pillow"
          }
        ]
      }]
}
```

Example of the Image Classification Model Configuration File

The following code uses the TensorFlow engine as an example. You can modify the **model_type** parameter based on the actual engine type.

- Model input

  Key: images

  Value: image files

- Model output

```
{
  "predicted_label": "flower",
  "scores": [
    ["rose", 0.99],
    ["begonia", 0.01]
```

```
        ]
    }
```

- Configuration file

```
{
    "model_type": "TensorFlow",
    "model_algorithm": "image_classification",
    "metrics": {
        "f1": 0.345294,
        "accuracy": 0.462963,
        "precision": 0.338977,
        "recall": 0.351852
    },
    "apis": [{
        "url": "/",
        "method": "post",
        "request": {
            "Content-type": "multipart/form-data",
            "data": {
                "type": "object",
                "properties": {
                    "images": {
                        "type": "file"
                    }
                }
            }
        },
        "response": {
            "Content-type": "application/json",
            "data": {
                "type": "object",
                "properties": {
                    "predicted_label": {
                        "type": "string"
                    },
                    "scores": {
                        "type": "array",
                        "items": [{
                            "type": "array",
                            "minItems": 2,
                            "maxItems": 2,
                            "items": [
                                {
                                    "type": "string"
                                },
                                {
                                    "type": "number"
                                }
                            ]
                        }]
                    }
                }
            }
        }
    }],
    "dependencies": [{
        "installer": "pip",
        "packages": [{
                "restraint": "ATLEAST",
                "package_version": "1.15.0",
                "package_name": "numpy"
            },
            {
                "restraint": "",
                "package_version": "",
                "package_name": "Pillow"
            }
        ]
    }]
}
```

The following code uses the MindSpore engine as an example. You can modify the **model_type** parameter based on the type of the engine you use.

- Model input

  Key: **images**

  Value: **image files**

- Model output
  ```
  "[[-2.404526  -3.0476532  -1.9888215  0.45013925 -1.7018927  0.40332815\n -7.1861157
  11.290332  -1.5861531  5.7887416 ]]"
  ```

- Configuration file
  ```json
  {
      "model_algorithm": "image_classification",
      "model_type": "MindSpore",
      "metrics": {
          "f1": 0.124555,
          "recall": 0.171875,
          "precision": 0.0023493892851938493,
          "accuracy": 0.00746268656716417
      },
      "apis": [{
          "url": "/",
          "method": "post",
          "request": {
              "Content-type": "multipart/form-data",
              "data": {
                  "type": "object",
                  "properties": {
                      "images": {
                          "type": "file"
                      }
                  }
              }
          },
          "response": {
              "Content-type": "applicaton/json",
              "data": {
                  "type": "object",
                  "properties": {
                      "mnist_result": {
                          "type": "array",
                          "item": [{
                              "type": "string"
                          }]
                      }
                  }
              }
          }
      }
      ],
      "dependencies": []
  }
  ```

Example of the Predictive Analytics Model Configuration File

The following code uses the TensorFlow engine as an example. You can modify the **model_type** parameter based on the actual engine type.

- Model input
  ```json
  {
      "data": {
          "req_data": [
              {
                  "buying_price": "high",
                  "maint_price": "high",
                  "doors": "2",
  ```

```
                "persons": "2",
                "lug_boot": "small",
                "safety": "low",
                "acceptability": "acc"
            },
            {
                "buying_price": "high",
                "maint_price": "high",
                "doors": "2",
                "persons": "2",
                "lug_boot": "small",
                "safety": "low",
                "acceptability": "acc"
            }
        ]
    }
}
```

- Model output

```
{
    "data": {
        "resp_data": [
            {
                "predict_result": "unacc"
            },
            {
                "predict_result": "unacc"
            }
        ]
    }
}
```

- Configuration file

  □ NOTE

  In the code, the **data** parameter in the request and response structures is described in JSON Schema. The content in **data** and **properties** corresponds to the model input and output.

```
{
    "model_type": "TensorFlow",
    "model_algorithm": "predict_analysis",
    "metrics": {
        "f1": 0.345294,
        "accuracy": 0.462963,
        "precision": 0.338977,
        "recall": 0.351852
    },
    "apis": [
        {
            "url": "/",
            "method": "post",
            "request": {
                "Content-type": "application/json",
                "data": {
                    "type": "object",
                    "properties": {
                        "data": {
                            "type": "object",
                            "properties": {
                                "req_data": {
                                    "items": [
                                        {
                                            "type": "object",
                                            "properties": {}
                                        }
                                    ],
                                    "type": "array"
                                }
                            }
```

```
                    }
                  }
                }
              },
              "response": {
                "Content-type": "application/json",
                "data": {
                  "type": "object",
                  "properties": {
                    "data": {
                      "type": "object",
                      "properties": {
                        "resp_data": {
                          "type": "array",
                          "items": [
                            {
                              "type": "object",
                              "properties": {}
                            }
                          ]
                        }
                      }
                    }
                  }
                }
              }
            }
          ],
          "dependencies": [
            {
              "installer": "pip",
              "packages": [
                {
                  "restraint": "EXACT",
                  "package_version": "1.15.0",
                  "package_name": "numpy"
                },
                {
                  "restraint": "EXACT",
                  "package_version": "5.2.0",
                  "package_name": "Pillow"
                }
              ]
            }
          ]
        }
```

Example of the Custom Image Model Configuration File

The model input and output are similar to those in **Example of the Object Detection Model Configuration File**.

- If the input is an image, the request example is as follows.

  In the example, a model prediction request containing the parameter **images** with the parameter type of **file** is received. For this example, the file upload button is displayed on the inference page, and the inference is performed in file format.

```
{
  "Content-type": "multipart/form-data",
  "data": {
    "type": "object",
    "properties": {
      "images": {
        "type": "file"
      }
    }
  }
```

```
        }
    }
```

- If the input is JSON data, the request example is as follows.

  In this example, the model prediction JSON request body is received. In the request, there is only one prediction request containing the parameter **input** with the parameter type of string. On the inference page, a text box is displayed for you to enter the prediction request.

```
{
    "Content-type": "application/json",
    "data": {
        "type": "object",
        "properties": {
            "input": {
                "type": "string"
            }
        }
    }
}
```

A complete request example is as follows:

```
{
    "model_algorithm": "image_classification",
    "model_type": "Image",
    "metrics": {
        "f1": 0.345294,
        "accuracy": 0.462963,
        "precision": 0.338977,
        "recall": 0.351852
    },
    "apis": [{
        "url": "/",
        "method": "post",
        "request": {
            "Content-type": "multipart/form-data",
            "data": {
                "type": "object",
                "properties": {
                    "images": {
                        "type": "file"
                    }
                }
            }
        },
        "response": {
            "Content-type": "application/json",
            "data": {
                "type": "object",
                "required": [
                    "predicted_label",
                    "scores"
                ],
                "properties": {
                    "predicted_label": {
                        "type": "string"
                    },
                    "scores": {
                        "type": "array",
                        "items": [{
                            "type": "array",
                            "minItems": 2,
                            "maxItems": 2,
                            "items": [{
                                    "type": "string"
                                },
                                {
                                    "type": "number"
```

```
                          }
                      ]
                  }]
              }
          }
      }
  }
  }]
}
```

Example of the Machine Learning Model Configuration File

The following uses XGBoost as an example:

- Model input

```
{
  "req_data": [
      {
          "sepal_length": 5,
          "sepal_width": 3.3,
          "petal_length": 1.4,
          "petal_width": 0.2
      },
      {
          "sepal_length": 5,
          "sepal_width": 2,
          "petal_length": 3.5,
          "petal_width": 1
      },
      {
          "sepal_length": 6,
          "sepal_width": 2.2,
          "petal_length": 5,
          "petal_width": 1.5
      }
  ]
}
```

- Model output

```
{
  "resp_data": [
      {
          "predict_result": "Iris-setosa"
      },
      {
          "predict_result": "Iris-versicolor"
      }
  ]
}
```

- Configuration file

```
{
  "model_type": "XGBoost",
  "model_algorithm": "xgboost_iris_test",
  "runtime": "python2.7",
  "metrics": {
      "f1": 0.345294,
      "accuracy": 0.462963,
      "precision": 0.338977,
      "recall": 0.351852
  },
  "apis": [
      {
          "url": "/",
          "method": "post",
          "request": {
              "Content-type": "application/json",
              "data": {
```

```
                        "type": "object",
                        "properties": {
                            "req_data": {
                                "items": [
                                    {
                                        "type": "object",
                                        "properties": {}
                                    }
                                ],
                                "type": "array"
                            }
                        }
                    }
                },
                "response": {
                    "Content-type": "applicaton/json",
                    "data": {
                        "type": "object",
                        "properties": {
                            "resp_data": {
                                "type": "array",
                                "items": [
                                    {
                                        "type": "object",
                                        "properties": {
                                            "predict_result": {}
                                        }
                                    }
                                ]
                            }
                        }
                    }
                }
            }
        ]
    }
```

Example of a Model Configuration File Using a Custom Dependency Package

The following example defines the NumPy 1.16.4 dependency environment.

```
{
    "model_algorithm": "image_classification",
    "model_type": "TensorFlow",
    "runtime": "python3.6",
    "apis": [
        {
            "url": "/",
            "method": "post",
            "request": {
                "Content-type": "multipart/form-data",
                "data": {
                    "type": "object",
                    "properties": {
                        "images": {
                            "type": "file"
                        }
                    }
                }
            },
            "response": {
                "Content-type": "applicaton/json",
                "data": {
                    "type": "object",
                    "properties": {
                        "mnist_result": {
                            "type": "array",
                            "item": [
```

```
                    {
                      "type": "string"
                    }
                  ]
              }
            }
          }
        }
      }
    ],
    "metrics": {
      "f1": 0.124555,
      "recall": 0.171875,
      "precision": 0.00234938928519385,
      "accuracy": 0.00746268656716417
    },
    "dependencies": [
      {
        "installer": "pip",
        "packages": [
          {
            "restraint": "EXACT",
            "package_version": "1.16.4",
            "package_name": "numpy"
          }
        ]
      }
    ]
}
```

4.1.3 Specifications for Writing Model Inference Code

This section describes the general method of editing model inference code in ModelArts. For details about the custom script examples (including inference code examples) of mainstream AI engines, see **Examples of Custom Scripts**. This section also provides an inference code example for the TensorFlow engine and an example of customizing the inference logic in the inference script.

Due to the limitation of API Gateway, the duration of a single prediction in ModelArts cannot exceed 40s. The model inference code must be logically clear and concise for satisfactory inference performance.

Specifications for Compiling Inference Code

1. In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see **Table 4-9**. The Python packages related to import statements have been configured in the ModelArts environment. You do not need to install them.

Table 4-9 Import statements of different types of parent model classes

Model Type	Parent Class	Import Statement
TensorFlow	TfServingBaseService	from model_service.tfserving_model_service import TfServingBaseService
PyTorch	PTServingBaseService	from model_service.pytorch_model_service import PTServingBaseService

Model Type	Parent Class	Import Statement
MindSpore	SingleNodeService	from model_service.model_service import SingleNodeService

2. The following methods can be rewritten:

Table 4-10 Methods to be rewritten

Method	Description
__init__(self, model_name, model_path)	Initialization method, which is suitable for models created based on deep learning frameworks. Models and labels are loaded using this method. This method must be rewritten for models based on PyTorch and Caffe to implement the model loading logic.
__init__(self, model_path)	Initialization method, which is suitable for models created based on machine learning frameworks. The model path (**self.model_path**) is initialized using this method. In Spark_MLlib, this method also initializes SparkSession (**self.spark**).
_preprocess(self, data)	Preprocess method, which is called before an inference request and is used to convert the original request data of an API into the expected input data of a model
_inference(self, data)	Inference request method. You are advised not to rewrite the method because once the method is rewritten, the built-in inference process of ModelArts will be overwritten and the custom inference logic will run.
_postprocess(self, data)	Postprocess method, which is called after an inference request is complete and is used to convert the model output to the API output

◫ NOTE

- You can choose to rewrite the preprocess and postprocess methods to implement preprocessing of the API input and postprocessing of the inference output.
- Rewriting the init method of the parent model class may cause an AI application to run abnormally.

3. The attribute that can be used is the local path where the model resides. The attribute name is **self.model_path**. In addition, PySpark-based models can use **self.spark** to obtain the SparkSession object in **customize_service.py**.

📖 NOTE

An absolute path is required for reading files in the inference code. You can obtain the local path of the model from the **self.model_path** attribute.

- When TensorFlow, Caffe, or MXNet is used, **self.model_path** indicates the path of the model file. See the following example:
  ```
  # Store the label.json file in the model directory. The following information is read:
  with open(os.path.join(self.model_path, 'label.json')) as f:
      self.label = json.load(f)
  ```

- When PyTorch, Scikit_Learn, or PySpark is used, **self.model_path** indicates the path of the model file. See the following example:
  ```
  # Store the label.json file in the model directory. The following information is read:
  dir_path = os.path.dirname(os.path.realpath(self.model_path))
  with open(os.path.join(dir_path, 'label.json')) as f:
      self.label = json.load(f)
  ```

4. **data** imported through the API for pre-processing, actual inference request, and post-processing can be **multipart/form-data** or **application/json**.

   – **multipart/form-data** request
   ```
   curl -X POST \
    <modelarts-inference-endpoint> \
    -F image1=@cat.jpg \
    -F images2=@horse.jpg
   ```

   The corresponding input data is as follows:

   ```
   [
     {
       "image1":{
         "cat.jpg":"<cat.jpg file io>"
       }
     },
     {
       "image2":{
         "horse.jpg":"<horse.jpg file io>"
       }
     }
   ]
   ```

   – **application/json** request
   ```
   curl -X POST \
    <modelarts-inference-endpoint> \
    -d '{
    "images":"base64 encode image"
     }'
   ```

   The corresponding input data is **python dict**.

   ```
   {
     "images":"base64 encode image"
   }
   ```

TensorFlow Inference Script Example

The following is an example of TensorFlow MnistService. For more TensorFlow inference code examples, see **TensorFlow** and **TensorFlow 2.1**.For details about the inference code of other engines, see **PyTorch** and **Caffe**.

- Inference code
  ```
  from PIL import Image
  import numpy as np
  from model_service.tfserving_model_service import TfServingBaseService

  class MnistService(TfServingBaseService):

      def _preprocess(self, data):
          preprocessed_data = {}
  ```

```
        for k, v in data.items():
            for file_name, file_content in v.items():
                image1 = Image.open(file_content)
                image1 = np.array(image1, dtype=np.float32)
                image1.resize((1, 784))
                preprocessed_data[k] = image1

        return preprocessed_data

    def _postprocess(self, data):

        infer_output = {}

        for output_name, result in data.items():

            infer_output["mnist_result"] = result[0].index(max(result[0]))

        return infer_output
```

- Request
  ```
  curl -X POST \ Real-time service address \ -F images=@test.jpg
  ```
- Response
  ```
  {"mnist_result": 7}
  ```

The preceding code example resizes images imported to the user's form to adapt to the model input shape. The **32×32** image is read from the Pillow library and resized to **1×784** to match the model input. In subsequent processing, convert the model output into a list for the RESTful API to display.

XGBoost Inference Script Example

For details about the inference code of other machine learning engines, see **PySpark** and **Scikit-learn**.

```python
# coding:utf-8
import collections
import json
import xgboost as xgb
from model_service.python_model_service import XgSklServingBaseService


class UserService(XgSklServingBaseService):

    # request data preprocess
    def _preprocess(self, data):
        list_data = []
        json_data = json.loads(data, object_pairs_hook=collections.OrderedDict)
        for element in json_data["data"]["req_data"]:
            array = []
            for each in element:
                array.append(element[each])
            list_data.append(array)
        return list_data

    #  predict
    def _inference(self, data):
        xg_model = xgb.Booster(model_file=self.model_path)
        pre_data = xgb.DMatrix(data)
        pre_result = xg_model.predict(pre_data)
        pre_result = pre_result.tolist()
        return pre_result

    # predict result process
    def _postprocess(self, data):
        resp_data = []
        for element in data:
```

```
        resp_data.append({"predict_result": element})
        return resp_data
```

Inference Script Example of the Custom Inference Logic

Customize a dependency package in the configuration file by referring to **Example of a Model Configuration File Using a Custom Dependency Package**. Then, use the following code example to load the model in **saved_model** format for inference.

📖 **NOTE**

> The logging module of Python used by the base inference image uses the default log level Warning. Only warning logs can be queried by default. To query INFO logs, set the log level to INFO in the code.

```python
# -*- coding: utf-8 -*-
import json
import os
import threading
import numpy as np
import tensorflow as tf
from PIL import Image
from model_service.tfserving_model_service import TfServingBaseService
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

class MnistService(TfServingBaseService):
    def __init__(self, model_name, model_path):
        self.model_name = model_name
        self.model_path = model_path
        self.model_inputs = {}
        self.model_outputs = {}

        # The label file can be loaded here and used in the post-processing function.
        # Directories for storing the label.txt file on OBS and in the model package

        # with open(os.path.join(self.model_path, 'label.txt')) as f:
        #     self.label = json.load(f)

        # Load the model in saved_model format in non-blocking mode to prevent blocking timeout.
        thread = threading.Thread(target=self.get_tf_sess)
        thread.start()

    def get_tf_sess(self):
        # Load the model in saved_model format.
        # The session will be reused. Do not use the with statement.
        sess = tf.Session(graph=tf.Graph())
        meta_graph_def = tf.saved_model.loader.load(sess, [tf.saved_model.tag_constants.SERVING], self.model_path)
        signature_defs = meta_graph_def.signature_def
        self.sess = sess
        signature = []

        # only one signature allowed
        for signature_def in signature_defs:
            signature.append(signature_def)
        if len(signature) == 1:
            model_signature = signature[0]
        else:
            logger.warning("signatures more than one, use serving_default signature")
            model_signature = tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY

        logger.info("model signature: %s", model_signature)

        for signature_name in meta_graph_def.signature_def[model_signature].inputs:
```

```
            tensorinfo = meta_graph_def.signature_def[model_signature].inputs[signature_name]
            name = tensorinfo.name
            op = self.sess.graph.get_tensor_by_name(name)
            self.model_inputs[signature_name] = op

        logger.info("model inputs: %s", self.model_inputs)

        for signature_name in meta_graph_def.signature_def[model_signature].outputs:
            tensorinfo = meta_graph_def.signature_def[model_signature].outputs[signature_name]
            name = tensorinfo.name
            op = self.sess.graph.get_tensor_by_name(name)
            self.model_outputs[signature_name] = op

        logger.info("model outputs: %s", self.model_outputs)

    def _preprocess(self, data):
        # Two request modes using HTTPS
        # 1. The request in form-data file format is as follows: data = {"Request key value":{"File
name":<File io>}}
        # 2. Request in JSON format is as follows: data = json.loads("JSON body transferred by the API")
        preprocessed_data = {}

        for k, v in data.items():
            for file_name, file_content in v.items():
                image1 = Image.open(file_content)
                image1 = np.array(image1, dtype=np.float32)
                image1.resize((1, 28, 28))
                preprocessed_data[k] = image1

        return preprocessed_data

    def _inference(self, data):
        feed_dict = {}
        for k, v in data.items():
            if k not in self.model_inputs.keys():
                logger.error("input key %s is not in model inputs %s", k, list(self.model_inputs.keys()))
                raise Exception("input key %s is not in model inputs %s" % (k, list(self.model_inputs.keys())))
            feed_dict[self.model_inputs[k]] = v

        result = self.sess.run(self.model_outputs, feed_dict=feed_dict)
        logger.info('predict result : ' + str(result))
        return result

    def _postprocess(self, data):
        infer_output = {"mnist_result": []}
        for output_name, results in data.items():

            for result in results:
                infer_output["mnist_result"].append(np.argmax(result))

        return infer_output

    def __del__(self):
        self.sess.close()
```

> ☐ NOTE
>
> To load models that are not supported by ModelArts or multiple models, specify the loading path using the **__init__** method. Example code:

```
# -*- coding: utf-8 -*-
import os
from model_service.tfserving_model_service import TfServingBaseService

class MnistService(TfServingBaseService):
    def __init__(self, model_name, model_path):
        # Obtain the path to the model folder.
        root = os.path.dirname(os.path.abspath(__file__))
        # test.onnx is the name of the model file to be loaded and must be stored in the model folder.
        self.model_path = os.path.join(root, test.onnx)

        # Loading multiple models, for example, test2.onnx
        # self.model_path2 = os.path.join(root, test2.onnx)
```

4.2 Model Templates

4.2.1 Introduction to Model Templates

> ☐ NOTE
>
> Importing a model from a template will be unavailable soon. After it goes offline, you can use the templates for AI engine and model configurations by choosing **OBS**, setting **AI Engine** to **Custom**, and importing your custom AI engine.

Because the configurations of models with the same functions are similar, ModelArts integrates the configurations of such models into a common template. By using this template, you can easily and quickly import models and create AI applications without writing the **config.json** configuration file. In simple terms, a template integrates AI engine and model configurations. Each template corresponds to a specific AI engine and inference mode. With the templates, you can quickly import models to ModelArts and create AI applications.

Background

Templates include general and non-general templates.

- Non-general templates are customized for specific scenarios with the input and output mode fixed. For example, the **TensorFlow-based image classification template** uses the built-in image processing mode.

- General templates integrate a specific AI engine and running environment and use the undefined input and output mode. Select an input and output mode based on the model function or application scenario to overwrite the undefined mode. For example, an image classification model requires the built-in image processing mode, and an object detection model requires the built-in object detection mode.

  > ☐ NOTE
  >
  > The models imported in undefined mode cannot be deployed as batch services.

Using a Template

Upload the model package to OBS before using the template. Store the model files in the **model** directory. When creating an AI application using this template, select the **model** directory. For details, see **Importing a Meta Model from a Template**.

Supported Templates

- **TensorFlow-based Image Classification Template**
- **TensorFlow-py27 General Template**
- **TensorFlow-py36 General Template**
- **MXNet-py27 General Template**
- **MXNet-py36 General Template**
- **PyTorch-py27 General Template**
- **PyTorch-py36 General Template**
- **Caffe-CPU-py27 General Template**
- **Caffe-GPU-py27 General Template**
- **Caffe-CPU-py36 General Template**
- **Caffe-GPU-py36 General Template**
- **Arm-Ascend Template**

Supported Input and Output Modes

- **Built-in Object Detection Mode**
- **Built-in Image Processing Mode**
- **Built-in Predictive Analytics Mode**
- **Undefined Mode**

4.2.2 Templates

4.2.2.1 TensorFlow-based Image Classification Template

Introduction

AI engine: TensorFlow 1.8; Environment: Python 2.7. This template is used to import a TensorFlow-based image classification model saved in **SavedModel** format. This template uses the built-in image processing mode of ModelArts. For details about the image processing mode, see **Built-in Image Processing Mode**. Ensure that your model can process images whose **key** is **images**, because you need to input an image whose **key** is **images** to the model for inference. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the TensorFlow-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

The built-in image processing mode cannot be overwritten. You cannot select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For details, see the model package example.
├── Custom Python package       //(Optional) User's Python package, which can be directly referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the TensorFlow-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>    (Optional) User's Python package, which can be directly referenced in model inference code
    ├── saved_model.pb        (Mandatory) Protocol buffer file, which contains the diagram description of the model
    ├── variables           Mandatory for the main file of the *.pb model. The folder must be named variables and contains the weight deviation of the model.
        ├── variables.index              Mandatory
        ├── variables.data-00000-of-00001      Mandatory
    ├──customize_service.py   (Mandatory) Model inference code file. The file must be named customize_service.py. Only one inference code file exists. The .py file on which customize_service.py depends can be directly put in the model directory.
```

4.2.2.2 TensorFlow-py27 General Template

Introduction

AI engine: TensorFlow 1.8; Environment: python2.7; Input and output mode: undefined mode. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the TensorFlow-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file                //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package        //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the TensorFlow-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>    (Optional) User's Python package, which can be directly referenced
in model inference code
    ├── saved_model.pb        (Mandatory) Protocol buffer file, which contains the diagram description of
the model
    ├── variables           Mandatory for the main file of the *.pb model. The folder must be named
variables and contains the weight deviation of the model.
        ├── variables.index                Mandatory
        ├── variables.data-00000-of-00001      Mandatory
    ├──customize_service.py   (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.3 TensorFlow-py36 General Template

Introduction

AI engine: TensorFlow 1.8; Environment: Python 3.6; Input and output mode: undefined mode. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the TensorFlow-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:
  ```
  model/
  │
  ├── Model file              //(Mandatory) The model file format varies according to the engine. For
  details, see the model package example.
  ├── Custom Python package          //(Optional) User's Python package, which can be directly
  referenced in model inference code
  ├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
  customize_service.py. Otherwise, the code is not considered as inference code.
  ```

Model Package Example

Structure of the TensorFlow-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model   (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>   (Optional) User's Python package, which can be directly referenced
in model inference code
    ├── saved_model.pb        (Mandatory) Protocol buffer file, which contains the diagram description of
the model
    ├── variables          Mandatory for the main file of the *.pb model. The folder must be named
variables and contains the weight deviation of the model.
        ├── variables.index              Mandatory
        ├── variables.data-00000-of-00001      Mandatory
    ├──customize_service.py   (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.4 MXNet-py27 General Template

Introduction

AI engine: MXNet 1.2.1; Environment: Python 2.7; Input and output mode: undefined mode. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the MXNet-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
|
├── Model file              //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package        //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the MXNet-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>      (Optional) User's Python package, which can be directly
referenced in model inference code
    ├── resnet-50-symbol.json    (Mandatory) Model definition file, which contains the neural network
description of the model
    ├── resnet-50-0000.params     (Mandatory) Model variable parameter file, which contains parameter
and weight information
    ├──customize_service.py      (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.5 MXNet-py36 General Template

Introduction

AI engine: MXNet 1.2.1; Environment: Python 3.6; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the MXNet-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For details, see the model package example.
├── Custom Python package          //(Optional) User's Python package, which can be directly referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the MXNet-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
     ├── <<Custom Python package>>       (Optional) User's Python package, which can be directly referenced in model inference code
     ├── resnet-50-symbol.json     (Mandatory) Model definition file, which contains the neural network description of the model
     ├── resnet-50-0000.params     (Mandatory) Model variable parameter file, which contains parameter and weight information
     ├──customize_service.py       (Mandatory) Model inference code file. The file must be named customize_service.py. Only one inference code file exists. The .py file on which customize_service.py depends can be directly put in the model directory.
```

4.2.2.6 PyTorch-py27 General Template

Introduction

AI engine: PyTorch 1.0; Environment: Python 2.7; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the PyTorch-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.
- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .
- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package           //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the PyTorch-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model     (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>     (Optional) User's Python package, which can be directly referenced
in model inference code
    ├── resnet50.pth         (Mandatory) PyTorch model file, which contains variable and weight information
    ├──customize_service.py   (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.7 PyTorch-py36 General Template

Introduction

AI engine: PyTorch 1.0; Environment: Python 3.6; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the PyTorch-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file                //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package          //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the PyTorch-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model     (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>     (Optional) User's Python package, which can be directly referenced
in model inference code
    ├── resnet50.pth           (Mandatory) PyTorch model file, which contains variable and weight information
    ├──customize_service.py   (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.8 Caffe-CPU-py27 General Template

Introduction

AI engine: CPU-based Caffe 1.0; Environment: Python 2.7; Input and output mode: undefined mode. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the Caffe-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For details, see the model package example.
├── Custom Python package         //(Optional) User's Python package, which can be directly referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the Caffe-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model     (Mandatory) The folder must be named model and is used to store model-related files.
   ├── <<Custom Python package>>      (Optional) User's Python package, which can be directly referenced in model inference code
   ├── deploy.prototxt           (Mandatory) Caffe model file, which contains information such as the model network structure
   ├── resnet.caffemodel        (Mandatory) Caffe model file, which contains variable and weight information
   ├── customize_service.py     (Mandatory) Model inference code file. The file must be named customize_service.py. Only one inference code file exists. The .py file on which customize_service.py depends can be directly put in the model directory.
```

4.2.2.9 Caffe-GPU-py27 General Template

Introduction

AI engine: GPU-based Caffe 1.0; Environment: Python 2.7; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the Caffe-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model**

folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package       //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the Caffe-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model     (Mandatory) The folder must be named model and is used to store model-related files.
   ├── <<Custom Python package>>      (Optional) User's Python package, which can be directly referenced
in model inference code
   ├── deploy.prototxt         (Mandatory) Caffe model file, which contains information such as the model
network structure
   ├── resnet.caffemodel       (Mandatory) Caffe model file, which contains variable and weight information
   ├── customize_service.py     (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.10 Caffe-CPU-py36 General Template

Introduction

AI engine: CPU-based Caffe 1.0; Environment: Python 3.6; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the Caffe-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package        //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the Caffe-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
   ├── <<Custom Python package>>     (Optional) User's Python package, which can be directly referenced
in model inference code
   ├── deploy.prototxt        (Mandatory) Caffe model file, which contains information such as the model
network structure
   ├── resnet.caffemodel      (Mandatory) Caffe model file, which contains variable and weight information
   ├── customize_service.py    (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.11 Caffe-GPU-py36 General Template

Introduction

AI engine: GPU-based Caffe 1.0; Environment: Python 3.6; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the Caffe-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For
```

```
details, see the model package example.
├── Custom Python package        //(Optional) User's Python package, which can be directly
referenced in model inference code
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the Caffe-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
   ├── <<Custom Python package>>     (Optional) User's Python package, which can be directly referenced
in model inference code
   ├── deploy.prototxt        (Mandatory) Caffe model file, which contains information such as the model
network structure
   ├── resnet.caffemodel      (Mandatory) Caffe model file, which contains variable and weight information
   ├── customize_service.py    (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.2.12 Arm-Ascend Template

Introduction

AI engine: MindSpore; Environment: Python 3.5; Input and output mode: Undefined. Select an appropriate input and output mode based on the model function or application scenario. When using the template to import a model, select the **model** directory containing the model files.

Template Input

The template input is the OM-based model package stored on OBS. Ensure that the OBS directory you use and ModelArts are in the same region. For details about model package requirements, see **Model Package Example**.

Input and Output Mode

Undefined Mode can be overwritten. You can select another input and output mode during model creation.

Model Package Specifications

- The model package must be stored in the OBS folder named **model**. Model files and the model inference code file are stored in the **model** folder.

- The model inference code file is mandatory. The file name must be **customize_service.py**. Only one inference code file can exist in the **model** folder. For details about how to write model inference code, see **Specifications for Writing Model Inference Code** .

- The structure of the model package imported using the template is as follows:

```
model/
│
├── Model file              //(Mandatory) The model file format varies according to the engine. For
details, see the model package example.
├── Custom Python package        //(Optional) User's Python package, which can be directly
referenced in model inference code
```

```
├── customize_service.py  //(Mandatory) Model inference code file. The file name must be
customize_service.py. Otherwise, the code is not considered as inference code.
```

Model Package Example

Structure of the OM-based model package

When publishing the model, you only need to specify the **model** directory.

```
OBS bucket/directory name
├── model    (Mandatory) The folder must be named model and is used to store model-related files.
    ├── <<Custom Python package>>    (Optional) User's Python package, which can be directly referenced
in model inference code
    ├── model.om          (Mandatory) Protocol buffer file, which contains the diagram description of the
model
    ├──customize_service.py   (Mandatory) Model inference code file. The file must be named
customize_service.py. Only one inference code file exists. The .py file on which customize_service.py
depends can be directly put in the model directory.
```

4.2.3 Input and Output Modes

4.2.3.1 Built-in Object Detection Mode

Input

This is a built-in input and output mode for object detection. The models using this mode are identified as object detection models. The prediction request path is **/**, the request protocol is **HTTP**, the request method is **POST**, **Content-Type** is **multipart/form-data**, **key** is **images**, and **type** is **file**. Before selecting this mode, ensure that your model can process the input data whose **key** is **images**.

Output

The inference result is returned in JSON format. For details about the fields, see **Table 4-11**.

Table 4-11 Parameters

Field	Type	Description
detection_classes	String array	List of detected objects, for example, **["yunbao","cat"]**
detection_boxes	Float array	Coordinates of the bounding box, in the format of $[Y_{min}, X_{min}, Y_{max}, X_{max}]$
detection_scores	Float array	Confidence scores of detected objects, which are used to measure the detection accuracy

The **JSON Schema** of the inference result is as follows:

```
{
    "type": "object",
    "properties": {
```

```
                    "detection_classes": {
                        "items": {
                            "type": "string"
                        },
                        "type": "array"
                    },
                    "detection_boxes": {
                        "items": {
                            "minItems": 4,
                            "items": {
                                "type": "number"
                            },
                            "type": "array",
                            "maxItems": 4
                        },
                        "type": "array"
                    },
                    "detection_scores": {
                        "items": {
                            "type": "string"
                        },
                        "type": "array"
                    }
                }
            }
        }
```

Sample Request

In this mode, input an image to be processed in the inference request. The inference result is returned in JSON format. The following are examples:

- Performing prediction on the console

  On the **Prediction** tab page of the service details page, upload an image and click **Predict** to obtain the prediction result.

- Using Postman to call a RESTful API for prediction

  After a model is deployed as a service, you can obtain the API URL on the **Usage Guides** tab page of the service details page.

  - On the **Headers** tab page, set **Content-Type** to **multipart/form-data** and **X-Auth-Token** to the actual token obtained.

    Figure 4-1 Setting the request header

  - On the **Body** tab page, set the request body. Set **key** to **images**, select **File**, select the image to be processed, and click **send** to send your prediction request.

Figure 4-2 Setting the request body
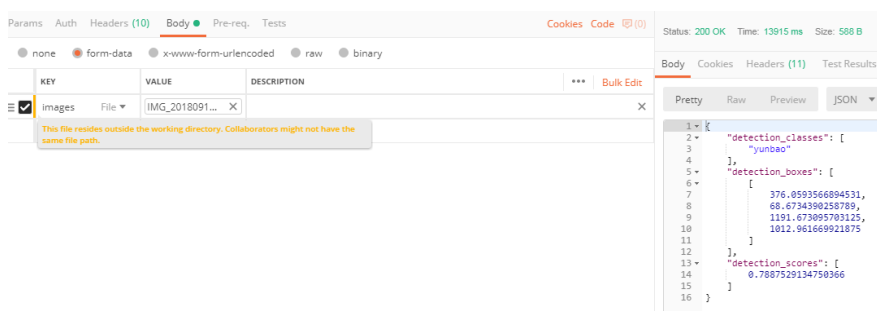
4.2.3.2 Built-in Image Processing Mode

Input

The built-in image processing input and output mode can be applied to models such as image classification, object detection, and image semantic segmentation. The prediction request path is **/**, the request protocol is **HTTPS**, the request method is **POST**, **Content-Type** is **multipart/form-data**, **key** is **images**, and **type** is **file**. Before selecting this mode, ensure that your model can process the input data whose **key** is **images**.

Output

The inference result is returned in JSON format. The specific fields are determined by the model.

Sample Request

In this mode, input an image to be processed in the inference request. The response in JSON format varies according to the model. The following are examples:

- Performing prediction on the console
- Using Postman to call a RESTful API for prediction

  After a model is deployed as a service, you can obtain the API URL on the **Usage Guides** tab page of the service details page. On the **Body** tab page, set the request body. Set **key** to **images**, select **File**, select the image to be processed, and click **send** to send your prediction request.

  Figure 4-3 Calling a RESTful API

4.2.3.3 Built-in Predictive Analytics Mode

Input

This is a built-in input and output mode for predictive analytics. The models using this mode are identified as predictive analytics models. The prediction request path is **/**, the request protocol is **HTTP**, the request method is **POST**, and **Content-Type** is **application/json**. The request body is in JSON format. For details about the JSON fields, see **Table 4-12**. Before selecting this mode, ensure that your model can process the input data in **JSON Schema** format. For details about the **JSON Schema** format, see the **official guide**.

Table 4-12 JSON field description

Field	Type	Description
data	Data structure	Inference data. For details, see **Table 4-13**.

Table 4-13 Data description

Field	Type	Description
req_data	ReqData array	List of inference data

ReqData is of the **Object** type and indicates the inference data. The data structure is determined by the application scenario. For models using this mode, the preprocessing logic in the custom model inference code should be able to correctly process the data inputted in the format defined by the mode.

The **JSON Schema** of a prediction request is as follows:

```
{
    "type": "object",
    "properties": {
        "data": {
            "type": "object",
            "properties": {
                "req_data": {
                    "items": [{
                        "type": "object",
                        "properties": {}
                    }],
                    "type": "array"
                }
            }
        }
    }
}
```

Output

The inference result is returned in JSON format. For details about the JSON fields, see **Table 4-14**.

Table 4-14 JSON field description

Field	Type	Description
data	Data structure	Inference data. For details, see **Table 4-15**.

Table 4-15 Data description

Field	Type	Description
resp_data	RespData array	List of prediction results

Similar to **ReqData**, **RespData** is also of the **Object** type and indicates the prediction result. Its structure is determined by the application scenario. For models using this mode, the postprocessing logic in the custom model inference code should be able to correctly output data in the format defined by the mode.

The **JSON Schema** of a prediction result is as follows:

```
{
    "type": "object",
    "properties": {
        "data": {
            "type": "object",
            "properties": {
                "resp_data": {
                    "type": "array",
                    "items": [{
                        "type": "object",
                        "properties": {}
                    }]
                }
            }
        }
    }
}
```

Sample Request

In this mode, input the data to be predicted in JSON format. The prediction result is returned in JSON format. The following are examples:

- Performing prediction on the console

  On the **Prediction** tab page of the service details page, enter inference code and click **Predict** to obtain the prediction result.

- Using Postman to call a RESTful API for prediction

  After a model is deployed as a service, you can obtain the API URL on the **Usage Guides** tab page of the service details page.

  – On the **Headers** tab page, set **Content-Type** to **application/json** and **X-Auth-Token** to the actual token obtained.

Figure 4-4 Setting the request header for prediction

– On the **Body** tab page, edit the data to be predicted and click **send** to send your prediction request.

4.2.3.4 Undefined Mode

Description

The undefined mode does not define the input and output mode. The input and output mode is determined by the model. Select this mode only when the existing input and output mode is not applicable to the application scenario of the model. The models imported in undefined mode cannot be deployed as batch services. In addition, the service prediction page may not be displayed properly.

Input

No limit.

Output

No limit.

Sample Request

The undefined mode has no specific sample request because the input and output of the request are entirely determined by the model.

4.3 Examples of Custom Scripts

4.3.1 TensorFlow

There are two types of TensorFlow APIs, Keras and tf. They use different code for training and saving models, but the same code for inference.

Training a Model (Keras API)

```
from keras.models import Sequential
model = Sequential()
from keras.layers import Dense
import tensorflow as tf

# Import a training dataset.
mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

print(x_train.shape)
```

```
from keras.layers import Dense
from keras.models import Sequential
import keras
from keras.layers import Dense, Activation, Flatten, Dropout

# Define a model network.
model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(units=5120,activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(units=10, activation='softmax'))

# Define an optimizer and loss functions.
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.summary()
# Train the model.
model.fit(x_train, y_train, epochs=2)
# Evaluate the model.
model.evaluate(x_test, y_test)
```

Saving a Model (Keras API)

```
from keras import backend as K

# K.get_session().run(tf.global_variables_initializer())

# Define the inputs and outputs of the prediction API.
# The key values of the inputs and outputs dictionaries are used as the index keys for the input and output
tensors of the model.
 # The input and output definitions of the model must match the custom inference script.
predict_signature = tf.saved_model.signature_def_utils.predict_signature_def(
    inputs={"images" : model.input},
    outputs={"scores" : model.output}
)

# Define a save path.
builder = tf.saved_model.builder.SavedModelBuilder('./mnist_keras/')

builder.add_meta_graph_and_variables(

    sess = K.get_session(),
    # The tf.saved_model.tag_constants.SERVING tag needs to be defined for inference and deployment.
    tags=[tf.saved_model.tag_constants.SERVING],
    """
    signature_def_map: Only single items can exist, or the corresponding key needs to be defined as follows:
    tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY
    """
    signature_def_map={
        tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:
            predict_signature
    }

)
builder.save()
```

Training a Model (tf API)

```
from __future__ import print_function

import gzip
import os
import urllib

import numpy
```

```
import tensorflow as tf
from six.moves import urllib

# Training data is obtained from the Yann LeCun official website http://yann.lecun.com/exdb/mnist/.
SOURCE_URL = 'http://yann.lecun.com/exdb/mnist/'
TRAIN_IMAGES = 'train-images-idx3-ubyte.gz'
TRAIN_LABELS = 'train-labels-idx1-ubyte.gz'
TEST_IMAGES = 't10k-images-idx3-ubyte.gz'
TEST_LABELS = 't10k-labels-idx1-ubyte.gz'
VALIDATION_SIZE = 5000


def maybe_download(filename, work_directory):
    """Download the data from Yann's website, unless it's already here."""
    if not os.path.exists(work_directory):
        os.mkdir(work_directory)
    filepath = os.path.join(work_directory, filename)
    if not os.path.exists(filepath):
        filepath, _ = urllib.request.urlretrieve(SOURCE_URL + filename, filepath)
        statinfo = os.stat(filepath)
        print('Successfully downloaded %s %d bytes.' % (filename, statinfo.st_size))
    return filepath


def _read32(bytestream):
    dt = numpy.dtype(numpy.uint32).newbyteorder('>')
    return numpy.frombuffer(bytestream.read(4), dtype=dt)[0]


def extract_images(filename):
    """Extract the images into a 4D uint8 numpy array [index, y, x, depth]."""
    print('Extracting %s' % filename)
    with gzip.open(filename) as bytestream:
        magic = _read32(bytestream)
        if magic != 2051:
            raise ValueError(
                'Invalid magic number %d in MNIST image file: %s' %
                (magic, filename))
        num_images = _read32(bytestream)
        rows = _read32(bytestream)
        cols = _read32(bytestream)
        buf = bytestream.read(rows * cols * num_images)
        data = numpy.frombuffer(buf, dtype=numpy.uint8)
        data = data.reshape(num_images, rows, cols, 1)
        return data


def dense_to_one_hot(labels_dense, num_classes=10):
    """Convert class labels from scalars to one-hot vectors."""
    num_labels = labels_dense.shape[0]
    index_offset = numpy.arange(num_labels) * num_classes
    labels_one_hot = numpy.zeros((num_labels, num_classes))
    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
    return labels_one_hot


def extract_labels(filename, one_hot=False):
    """Extract the labels into a 1D uint8 numpy array [index]."""
    print('Extracting %s' % filename)
    with gzip.open(filename) as bytestream:
        magic = _read32(bytestream)
        if magic != 2049:
            raise ValueError(
                'Invalid magic number %d in MNIST label file: %s' %
                (magic, filename))
        num_items = _read32(bytestream)
        buf = bytestream.read(num_items)
        labels = numpy.frombuffer(buf, dtype=numpy.uint8)
        if one_hot:
```

```python
            return dense_to_one_hot(labels)
        return labels


class DataSet(object):
    """Class encompassing test, validation and training MNIST data set."""

    def __init__(self, images, labels, fake_data=False, one_hot=False):
        """Construct a DataSet. one_hot arg is used only if fake_data is true."""

        if fake_data:
            self._num_examples = 10000
            self.one_hot = one_hot
        else:
            assert images.shape[0] == labels.shape[0], (
                    'images.shape: %s labels.shape: %s' % (images.shape,
                                                 labels.shape))
            self._num_examples = images.shape[0]

            # Convert shape from [num examples, rows, columns, depth]
            # to [num examples, rows*columns] (assuming depth == 1)
            assert images.shape[3] == 1
            images = images.reshape(images.shape[0],
                             images.shape[1] * images.shape[2])
            # Convert from [0, 255] -> [0.0, 1.0].
            images = images.astype(numpy.float32)
            images = numpy.multiply(images, 1.0 / 255.0)
        self._images = images
        self._labels = labels
        self._epochs_completed = 0
        self._index_in_epoch = 0

    @property
    def images(self):
        return self._images

    @property
    def labels(self):
        return self._labels

    @property
    def num_examples(self):
        return self._num_examples

    @property
    def epochs_completed(self):
        return self._epochs_completed

    def next_batch(self, batch_size, fake_data=False):
        """Return the next `batch_size` examples from this data set."""
        if fake_data:
            fake_image = [1] * 784
            if self.one_hot:
                fake_label = [1] + [0] * 9
            else:
                fake_label = 0
            return [fake_image for _ in range(batch_size)], [
                fake_label for _ in range(batch_size)
            ]
        start = self._index_in_epoch
        self._index_in_epoch += batch_size
        if self._index_in_epoch > self._num_examples:
            # Finished epoch
            self._epochs_completed += 1
            # Shuffle the data
            perm = numpy.arange(self._num_examples)
            numpy.random.shuffle(perm)
            self._images = self._images[perm]
            self._labels = self._labels[perm]
```

```
            # Start next epoch
            start = 0
            self._index_in_epoch = batch_size
            assert batch_size <= self._num_examples
        end = self._index_in_epoch
        return self._images[start:end], self._labels[start:end]


def read_data_sets(train_dir, fake_data=False, one_hot=False):
    """Return training, validation and testing data sets."""

    class DataSets(object):
        pass

    data_sets = DataSets()

    if fake_data:
        data_sets.train = DataSet([], [], fake_data=True, one_hot=one_hot)
        data_sets.validation = DataSet([], [], fake_data=True, one_hot=one_hot)
        data_sets.test = DataSet([], [], fake_data=True, one_hot=one_hot)
        return data_sets

    local_file = maybe_download(TRAIN_IMAGES, train_dir)
    train_images = extract_images(local_file)

    local_file = maybe_download(TRAIN_LABELS, train_dir)
    train_labels = extract_labels(local_file, one_hot=one_hot)

    local_file = maybe_download(TEST_IMAGES, train_dir)
    test_images = extract_images(local_file)

    local_file = maybe_download(TEST_LABELS, train_dir)
    test_labels = extract_labels(local_file, one_hot=one_hot)

    validation_images = train_images[:VALIDATION_SIZE]
    validation_labels = train_labels[:VALIDATION_SIZE]
    train_images = train_images[VALIDATION_SIZE:]
    train_labels = train_labels[VALIDATION_SIZE:]

    data_sets.train = DataSet(train_images, train_labels)
    data_sets.validation = DataSet(validation_images, validation_labels)
    data_sets.test = DataSet(test_images, test_labels)
    return data_sets

training_iteration = 1000

modelarts_example_path = './modelarts-mnist-train-save-deploy-example'

export_path = modelarts_example_path + '/model/'
data_path = './'

print('Training model...')
mnist = read_data_sets(data_path, one_hot=True)
sess = tf.InteractiveSession()
serialized_tf_example = tf.placeholder(tf.string, name='tf_example')
feature_configs = {'x': tf.FixedLenFeature(shape=[784], dtype=tf.float32), }
tf_example = tf.parse_example(serialized_tf_example, feature_configs)
x = tf.identity(tf_example['x'], name='x')  # use tf.identity() to assign name
y_ = tf.placeholder('float', shape=[None, 10])
w = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
sess.run(tf.global_variables_initializer())
y = tf.nn.softmax(tf.matmul(x, w) + b, name='y')
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
values, indices = tf.nn.top_k(y, 10)
table = tf.contrib.lookup.index_to_string_table_from_tensor(
    tf.constant([str(i) for i in range(10)]))
prediction_classes = table.lookup(tf.to_int64(indices))
```

```
for _ in range(training_iteration):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, 'float'))
print('training accuracy %g' % sess.run(
    accuracy, feed_dict={
        x: mnist.test.images,
        y_: mnist.test.labels
    }))
print('Done training!')
```

Saving a Model (tf API)

```
# Export the model.
# The model needs to be saved using the saved_model API.
print('Exporting trained model to', export_path)
builder = tf.saved_model.builder.SavedModelBuilder(export_path)

tensor_info_x = tf.saved_model.utils.build_tensor_info(x)
tensor_info_y = tf.saved_model.utils.build_tensor_info(y)

# Define the inputs and outputs of the prediction API.
# The key values of the inputs and outputs dictionaries are used as the index keys for the input and output
tensors of the model.
 # The input and output definitions of the model must match the custom inference script.
prediction_signature = (
    tf.saved_model.signature_def_utils.build_signature_def(
        inputs={'images': tensor_info_x},
        outputs={'scores': tensor_info_y},
        method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NAME))

legacy_init_op = tf.group(tf.tables_initializer(), name='legacy_init_op')
builder.add_meta_graph_and_variables(
    # Set tag to serve/tf.saved_model.tag_constants.SERVING.
    sess, [tf.saved_model.tag_constants.SERVING],
    signature_def_map={
        'predict_images':
            prediction_signature,
    },
    legacy_init_op=legacy_init_op)

builder.save()

print('Done exporting!')
```

Inference Code (Keras and tf APIs)

In the model inference code file **customize_service.py**, add a child model class
which inherits properties from its parent model class. For details about the import
statements of different types of parent model classes, see **Table 4-9**.

```
from PIL import Image
import numpy as np
from model_service.tfserving_model_service import TfServingBaseService


class MnistService(TfServingBaseService):

    # Match the model input with the user's HTTPS API input during preprocessing.
    # The model input corresponding to the preceding training part is {"images":<array>}.
    def _preprocess(self, data):

        preprocessed_data = {}
        images = []
        # Iterate the input data.
        for k, v in data.items():
            for file_name, file_content in v.items():
```

```
            image1 = Image.open(file_content)
            image1 = np.array(image1, dtype=np.float32)
            image1.resize((1,784))
            images.append(image1)
        # Return the numpy array.
        images = np.array(images,dtype=np.float32)
        # Perform batch processing on multiple input samples and ensure that the shape is the same as that
inputted during training.
        images.resize((len(data), 784))
        preprocessed_data['images'] = images
        return preprocessed_data

    # Processing logic of the inference for invoking the parent class.

    # The output corresponding to model saving in the preceding training part is {"scores":<array>}.
    # Postprocess the HTTPS output.
    def _postprocess(self, data):
        infer_output = {"mnist_result": []}
        # Iterate the model output.
        for output_name, results in data.items():
            for result in results:
                infer_output["mnist_result"].append(result.index(max(result)))
        return infer_output
```

4.3.2 TensorFlow 2.1

Training and Saving a Model

```
from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  # Name the output layer output, which is used to obtain the result during model inference.
  tf.keras.layers.Dense(10, activation='softmax', name="output")
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10)

tf.keras.models.save_model(model, "./mnist")
```

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see **Table 4-9**.

```
import logging
import threading

import numpy as np
import tensorflow as tf
```

```python
from PIL import Image

from model_service.tfserving_model_service import TfServingBaseService

logger = logging.getLogger()
logger.setLevel(logging.INFO)


class MnistService(TfServingBaseService):

    def __init__(self, model_name, model_path):
        self.model_name = model_name
        self.model_path = model_path
        self.model = None
        self.predict = None

        # The label file can be loaded here and used in the post-processing function.
        # Directories for storing the label.txt file on OBS and in the model package

        # with open(os.path.join(self.model_path, 'label.txt')) as f:
        #     self.label = json.load(f)
        # Load the model in saved_model format in non-blocking mode to prevent blocking timeout.
        thread = threading.Thread(target=self.load_model)
        thread.start()

    def load_model(self):
        # Load the model in saved_model format.
        self.model = tf.saved_model.load(self.model_path)

        signature_defs = self.model.signatures.keys()

        signature = []
        # only one signature allowed
        for signature_def in signature_defs:
            signature.append(signature_def)

        if len(signature) == 1:
            model_signature = signature[0]
        else:
            logging.warning("signatures more than one, use serving_default signature from %s", signature)
            model_signature = tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY

        self.predict = self.model.signatures[model_signature]

    def _preprocess(self, data):
        images = []
        for k, v in data.items():
            for file_name, file_content in v.items():
                image1 = Image.open(file_content)
                image1 = np.array(image1, dtype=np.float32)
                image1.resize((28, 28, 1))
                images.append(image1)

        images = tf.convert_to_tensor(images, dtype=tf.dtypes.float32)
        preprocessed_data = images

        return preprocessed_data

    def _inference(self, data):

        return self.predict(data)

    def _postprocess(self, data):

        return {
            "result": int(data["output"].numpy()[0].argmax())
        }
```

4.3.3 PyTorch

Training a Model

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

# Define a network structure.
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
# The second dimension of the input must be 784.
        self.hidden1 = nn.Linear(784, 5120, bias=False)
        self.output = nn.Linear(5120, 10, bias=False)

    def forward(self, x):
        x = x.view(x.size()[0], -1)
        x = F.relu((self.hidden1(x)))
        x = F.dropout(x, 0.2)
        x = self.output(x)
        return F.log_softmax(x)

def train(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.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 10 == 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()))

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()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            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)))

device = torch.device("cpu")

batch_size=64

kwargs={}

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('.', train=True, download=True,
            transform=transforms.Compose([
```

```
            transforms.ToTensor()
        ])),
    batch_size=batch_size, shuffle=True, **kwargs)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('.', train=False, transform=transforms.Compose([
        transforms.ToTensor()
    ])),
    batch_size=1000, shuffle=True, **kwargs)

model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
optimizer = optim.Adam(model.parameters())

for epoch in range(1, 2 + 1):
    train(model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
```

Saving a Model

```
# The model must be saved using state_dict and can be deployed remotely.
torch.save(model.state_dict(), "pytorch_mnist/mnist_mlp.pt")
```

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see **Table 4-9**.

```
from PIL import Image
import log
from model_service.pytorch_model_service import PTServingBaseService
import torch.nn.functional as F

import torch.nn as nn
import torch
import json

import numpy as np

logger = log.getLogger(__name__)

import torchvision.transforms as transforms

# Define model preprocessing.
infer_transformation = transforms.Compose([
    transforms.Resize((28,28)),
    # Transform to a PyTorch tensor.
    transforms.ToTensor()
])


import os


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 tags.
        self.label = [0,1,2,3,4,5,6,7,8,9]
        # Labels can also be loaded by label file.
        # Store the label.json file in the model directory. The following information is read:
        dir_path = os.path.dirname(os.path.realpath(self.model_path))
```

```
        with open(os.path.join(dir_path, 'label.json')) as f:
            self.label = json.load(f)


    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

    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

    def _inference(self, data):

        result = {}
        for k, v in data.items():
            result[k] = self.model(v)

        return result

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden1 = nn.Linear(784, 5120, bias=False)
        self.output = nn.Linear(5120, 10, bias=False)

    def forward(self, x):
        x = x.view(x.size()[0], -1)
        x = F.relu((self.hidden1(x)))
        x = F.dropout(x, 0.2)
        x = self.output(x)
        return F.log_softmax(x)


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)
    # Declare an inference mode.
    model.eval()
```

```
    return model
```

4.3.4 Caffe

Training and Saving a Model

lenet_train_test.prototxt file

```
name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
```

```
          name: "pool1"
          type: "Pooling"
          bottom: "conv1"
          top: "pool1"
          pooling_param {
            pool: MAX
            kernel_size: 2
            stride: 2
          }
        }
        layer {
          name: "conv2"
          type: "Convolution"
          bottom: "pool1"
          top: "conv2"
          param {
            lr_mult: 1
          }
          param {
            lr_mult: 2
          }
          convolution_param {
            num_output: 50
            kernel_size: 5
            stride: 1
            weight_filler {
              type: "xavier"
            }
            bias_filler {
              type: "constant"
            }
          }
        }
        layer {
          name: "pool2"
          type: "Pooling"
          bottom: "conv2"
          top: "pool2"
          pooling_param {
            pool: MAX
            kernel_size: 2
            stride: 2
          }
        }
        layer {
          name: "ip1"
          type: "InnerProduct"
          bottom: "pool2"
          top: "ip1"
          param {
            lr_mult: 1
          }
          param {
            lr_mult: 2
          }
          inner_product_param {
            num_output: 500
            weight_filler {
              type: "xavier"
            }
            bias_filler {
              type: "constant"
            }
          }
        }
        layer {
          name: "relu1"
          type: "ReLU"
          bottom: "ip1"
```

```
    top: "ip1"
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

lenet_solver.prototxt file

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 1000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: CPU
```

Train the model.

```
./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt
```

The **caffemodel** file is generated after model training. Rewrite the **lenet_train_test.prototxt** file to the **lenet_deploy.prototxt** file used for deployment by modifying input and output layers.

```
name: "LeNet"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 1 dim: 1  dim: 28 dim: 28 } }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

```
    }
    layer {
      name: "pool2"
      type: "Pooling"
      bottom: "conv2"
      top: "pool2"
      pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
      }
    }
    layer {
      name: "ip1"
      type: "InnerProduct"
      bottom: "pool2"
      top: "ip1"
      param {
        lr_mult: 1
      }
      param {
        lr_mult: 2
      }
      inner_product_param {
        num_output: 500
        weight_filler {
          type: "xavier"
        }
        bias_filler {
          type: "constant"
        }
      }
    }
    layer {
      name: "relu1"
      type: "ReLU"
      bottom: "ip1"
      top: "ip1"
    }
    layer {
      name: "ip2"
      type: "InnerProduct"
      bottom: "ip1"
      top: "ip2"
      param {
        lr_mult: 1
      }
      param {
        lr_mult: 2
      }
      inner_product_param {
        num_output: 10
        weight_filler {
          type: "xavier"
        }
        bias_filler {
          type: "constant"
        }
      }
    }
    layer {
      name: "prob"
      type: "Softmax"
      bottom: "ip2"
      top: "prob"
    }
```

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see **Table 4-9**.

```
from model_service.caffe_model_service import CaffeBaseService

import numpy as np

import os, json

import caffe

from PIL import Image


class LenetService(CaffeBaseService):

  def __init__(self, model_name, model_path):
      # Call the inference method of the parent class.
      super(LenetService, self).__init__(model_name, model_path)

      # Configure preprocessing information.
      transformer = caffe.io.Transformer({'data': self.net.blobs['data'].data.shape})
      # Transform to NCHW.
      transformer.set_transpose('data', (2, 0, 1))
      # Perform normalization.
      transformer.set_raw_scale('data', 255.0)

      # If the batch size is set to 1, inference is supported for only one image.
      self.net.blobs['data'].reshape(1, 1, 28, 28)
      self.transformer = transformer

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


  def _preprocess(self, data):

      for k, v in data.items():
          for file_name, file_content in v.items():
              im = caffe.io.load_image(file_content, color=False)
              # Pre-process the images.
              self.net.blobs['data'].data[...] = self.transformer.preprocess('data', im)

              return

  def _postprocess(self, data):

      data = data['prob'][0, :]
      predicted = np.argmax(data)
      predicted = {"predicted" : str(predicted) }

      return predicted
```

4.3.5 XGBoost

Training and Saving a Model

```
import pandas as pd
import xgboost as xgb
from sklearn.model_selection import train_test_split

# Prepare training data and setting parameters
iris = pd.read_csv('/home/ma-user/work/iris.csv')
```

```
X = iris.drop(['variety'],axis=1)
y = iris[['variety']]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234565)
params = {
    'booster': 'gbtree',
    'objective': 'multi:softmax',
    'num_class': 3,
    'gamma': 0.1,
    'max_depth': 6,
    'lambda': 2,
    'subsample': 0.7,
    'colsample_bytree': 0.7,
    'min_child_weight': 3,
    'silent': 1,
    'eta': 0.1,
    'seed': 1000,
    'nthread': 4,
}
plst = params.items()
dtrain = xgb.DMatrix(X_train, y_train)
num_rounds = 500
model = xgb.train(plst, dtrain, num_rounds)
model.save_model('/tmp/xgboost.m')
```

Before training, download the **iris.csv** dataset, decompress it, and upload it to the **/home/ma-user/work/** directory of the notebook instance. Download the **iris.csv** dataset from **https://gist.github.com/netj/8836201**. For details about how to upload a file to a notebook instance, see **Upload Scenarios and Entries**.

After the model is saved, it must be uploaded to the OBS directory before being published. The **config.json** configuration and the **customize_service.py** inference code must be included during the publishing. For details about how to compile **config.json**, see **Specifications for Editing a Model Configuration File** . For details about inference code, see **Inference Code**.

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see **Table 4-9**.

```
# coding:utf-8
import collections
import json
import xgboost as xgb
from model_service.python_model_service import XgSklServingBaseService
class UserService(XgSklServingBaseService):

    # request data preprocess
    def _preprocess(self, data):
        list_data = []
        json_data = json.loads(data, object_pairs_hook=collections.OrderedDict)
        for element in json_data["data"]["req_data"]:
            array = []
            for each in element:
                array.append(element[each])
            list_data.append(array)
        return list_data

    #   predict
    def _inference(self, data):
        xg_model = xgb.Booster(model_file=self.model_path)
        pre_data = xgb.DMatrix(data)
        pre_result = xg_model.predict(pre_data)
```

```
        pre_result = pre_result.tolist()
        return pre_result

    # predict result process
    def _postprocess(self,data):
        resp_data = []
        for element in data:
            resp_data.append({"predictresult": element})
        return resp_data
```

4.3.6 PySpark

Training and Saving a Model

```
from pyspark.ml import Pipeline, PipelineModel
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import LogisticRegression

# Prepare training data using tuples.
# Prepare training data from a list of (label, features) tuples.
training = spark.createDataFrame([
    (1.0, Vectors.dense([0.0, 1.1, 0.1])),
    (0.0, Vectors.dense([2.0, 1.0, -1.0])),
    (0.0, Vectors.dense([2.0, 1.3, 1.0])),
    (1.0, Vectors.dense([0.0, 1.2, -0.5]))], ["label", "features"])

# Create a training instance. The logistic regression algorithm is used for training.
# Create a LogisticRegression instance. This instance is an Estimator.
lr = LogisticRegression(maxIter=10, regParam=0.01)

# Train the logistic regression model.
# Learn a LogisticRegression model. This uses the parameters stored in lr.
model = lr.fit(training)

# Save the model to a local directory.
# Save model to local path.
model.save("/tmp/spark_model")
```

After the model is saved, it must be uploaded to the OBS directory before being published. The **config.json** configuration and the **customize_service.py** inference code must be included during the publishing. For details about how to compile **config.json**, see **Specifications for Editing a Model Configuration File** . For details about inference code, see **Inference Code**.

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details about the import statements of different types of parent model classes, see **Table 4-9**.

```
# coding:utf-8
import collections
import json
import traceback

import model_service.log as log
from model_service.spark_model_service import SparkServingBaseService
from pyspark.ml.classification import LogisticRegression

logger = log.getLogger(__name__)


class UserService(SparkServingBaseService):
```

```
# Pre-process data.
def _preprocess(self, data):
    logger.info("Begin to handle data from user data...")
    # Read data.
    req_json = json.loads(data, object_pairs_hook=collections.OrderedDict)
    try:
        # Convert data to the spark dataframe format.
        predict_spdf = self.spark.createDataFrame(pd.DataFrame(req_json["data"]["req_data"]))
    except Exception as e:
        logger.error("check your request data does meet the requirements ?")
        logger.error(traceback.format_exc())
        raise Exception("check your request data does meet the requirements ?")
    return predict_spdf

# Perform model inference.
def _inference(self, data):
    try:
        # Load a model file.
        predict_model = LogisticRegression.load(self.model_path)
        # Perform data inference.
        prediction_result = predict_model.transform(data)
except Exception as e:
        logger.error(traceback.format_exc())
        raise Exception("Unable to load model and do dataframe transformation.")
    return prediction_result

# Post-process data.
def _postprocess(self, pre_data):
    logger.info("Get new data to respond...")
    predict_str = pre_data.toPandas().to_json(orient='records')
    predict_result = json.loads(predict_str)
    return predict_result
```

4.3.7 Scikit-learn

Training and Saving a Model

```
import json
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.externals import joblib
iris = pd.read_csv('/home/ma-user/work/iris.csv')
X = iris.drop(['variety'],axis=1)
y = iris[['variety']]
# Create a LogisticRegression instance and train model
logisticRegression = LogisticRegression(C=1000.0, random_state=0)
logisticRegression.fit(X,y)
# Save model to local path
joblib.dump(logisticRegression, '/tmp/sklearn.m')
```

Before training, download the **iris.csv** dataset, decompress it, and upload it to the **/home/ma-user/work/** directory of the notebook instance. Download the **iris.csv** dataset from **https://gist.github.com/netj/8836201**. For details about how to upload a file to a notebook instance, see **Upload Scenarios and Entries**.

After the model is saved, it must be uploaded to the OBS directory before being published. The **config.json** and **customize_service.py** files must be contained during publishing. For details about the definition method, see **Introduction to Model Package Specifications**.

Inference Code

In the model inference code file **customize_service.py**, add a child model class. This child model class inherits properties from its parent model class. For details

about the import statements of different types of parent model classes, see **Table 4-9**.

```python
# coding:utf-8
import collections
import json
from sklearn.externals import joblib
from model_service.python_model_service import XgSklServingBaseService

class UserService(XgSklServingBaseService):

    # request data preprocess
    def _preprocess(self, data):
        list_data = []
        json_data = json.loads(data, object_pairs_hook=collections.OrderedDict)
        for element in json_data["data"]["req_data"]:
            array = []
            for each in element:
                array.append(element[each])
                list_data.append(array)
        return list_data

    # predict
    def _inference(self, data):
        sk_model = joblib.load(self.model_path)
        pre_result = sk_model.predict(data)
        pre_result = pre_result.tolist()
        return pre_result

    # predict result process
    def _postprocess(self,data):
        resp_data = []
        for element in data:
            resp_data.append({"predictresult": element})
        return resp_data
```

5 ModelArts Monitoring on Cloud Eye

[ModelArts Metrics](#)

[Setting Alarm Rules](#)

[Viewing Monitoring Metrics](#)

5.1 ModelArts Metrics

Description

The cloud service platform provides Cloud Eye to help you better understand the status of your ModelArts real-time services and models. You can use Cloud Eye to automatically monitor your ModelArts real-time services and model loads in real time and manage alarms and notifications so that you can obtain the performance metrics of ModelArts and models.

Namespace

SYS.ModelArts

Monitoring Metrics

Table 5-1 ModelArts metrics

Metric ID	Metric Name	Description	Value Range	Monitored Entity	Monitoring Interval
cpu_usage	CPU Usage	CPU usage of ModelArts Unit: %	≥ 0%	ModelArts model loads	1 minute
mem_usage	Memory Usage	Memory usage of ModelArts Unit: %	≥ 0%	ModelArts model loads	1 minute

Metric ID	Metric Name	Description	Value Range	Monitored Entity	Monitoring Interval
gpu_util	GPU Usage	GPU usage of ModelArts Unit: %	≥ 0%	ModelArts model loads	1 minute
gpu_mem_usage	GPU Memory Usage	GPU memory usage of ModelArts Unit: %	≥ 0%	ModelArts model loads	1 minute
npu_util	NPU Usage	NPU usage of ModelArts Unit: %	≥ 0%	ModelArts model loads	1 minute
npu_mem_usage	NPU Memory Usage	NPU memory usage of ModelArts Unit: %	≥ 0%	ModelArts model loads	1 minute
successfully_called_times	Number of Successful Calls	Times that ModelArts has been successfully called Unit: times/minute	≥ counts/minute	ModelArts models ModelArts real-time services	1 minute
failed_called_times	Number of Failed Calls	Times that ModelArts failed to be called Unit: times/minute	≥ counts/minute	ModelArts models ModelArts real-time services	1 minute
total_called_times	Total Calls	Times that ModelArts is called Unit: times/minute	≥ counts/minute	ModelArts model loads ModelArts real-time services	1 minute
disk_read_rate	Disk Read Rate	Disk read rate of ModelArts Unit: bit/minute	≥ bit/minute	ModelArts model loads	1 minute
disk_write_rate	Disk Write Rate	Disk write rate of ModelArts Unit: bit/minute	≥ bit/minute	ModelArts model loads	1 minute

Metric ID	Metric Name	Description	Value Range	Monitored Entity	Monitoring Interval
send_bytes_rate	Uplink rate	Outbound traffic rate of ModelArts Unit: bit/minute	≥ bit/minute	ModelArts model loads	1 minute
recv_bytes_rate	Downlink rate	Inbound traffic rate of ModelArts	≥ bit/minute	ModelArts model loads	1 minute
req_count_2xx	2xx Responses	Number of times that the API returns a 2xx response	≥ counts/minute	ModelArts real-time services	1 minute
req_count_4xx	4xx Errors	Number of times that the API returns a 4xx error	≥ counts/minute	ModelArts real-time services	1 minute
req_count_5xx	5xx Errors	Number of times that the API returns a 5xx error	≥ counts/minute	ModelArts real-time services	1 minute
avg_latency	Average Latency	Average latency of the API	≥ ms	ModelArts real-time services	1 minute

If a measurement object has multiple measurement dimensions, all the measurement dimensions are mandatory when you use an API to query monitoring metrics.

- The following provides an example of using the multi-dimensional **dim** to query a single monitoring metric:
  dim.0=service_id,530cd6b0-86d7-4818-837f-935f6a27414d&dim.1="model_id, 3773b058-5b4f-4366-9035-9bbd9964714a

- The following provides an example of using the multi-dimensional **dim** to query monitoring metrics in batches:
  "dimensions": [

  {

  "name": "service_id",

  "value": "530cd6b0-86d7-4818-837f-935f6a27414d"

  }

  {

  "name": "model_id",

  "value": "3773b058-5b4f-4366-9035-9bbd9964714a"

  }

  ]

Dimensions

Table 5-2 Dimension description

Key	Value
service_id	Real-time service ID
model_id	Model ID

5.2 Setting Alarm Rules

Scenario

Setting alarm rules allows you to customize the monitored objects and notification policies so that you can know the status of ModelArts real-time services and models in a timely manner.

An alarm rule includes the alarm rule name, monitored object, metric, threshold, monitoring interval, and whether to send a notification. This section describes how to set alarm rules for ModelArts services and models.

◯ **NOTE**

Only real-time services in the **Running** status can be interconnected with CES.

Prerequisites

- A ModelArts real-time service has been created.
- ModelArts monitoring has been enabled on Cloud Eye. To do so, log in to the Cloud Eye console. On the Cloud Eye page, click **Custom Monitoring**. Then, enable ModelArts monitoring as prompted.

Procedure

Set an alarm rule in any of the following ways:

- Set an alarm rule for all ModelArts services.
- Set an alarm rule for a ModelArts service.
- Set an alarm rule for a model version.
- Set an alarm rule for a metric of a service or model version.

Method 1: Setting an Alarm Rule for All ModelArts Services

1. Log in to the management console.
2. On the **Service List**, click **Cloud Eye** under **Management & Governance**.
3. In the navigation pane on the left, choose **Alarm Management** > **Alarm Rules** and click **Create Alarm Rule**.

4. On the **Create Alarm Rule** page, set **Resource Type** to **ModelArts**, **Dimension** to **Service**, and **Method** to **Configure manually**, and set alarm policies. Then, confirm settings and click **Create**.

Method 2: Setting an Alarm Rule for a Single Service

1. Log in to the management console.

2. On the **Service List**, click **Cloud Eye** under **Management & Governance**.

3. In the left navigation pane, choose **Cloud Service Monitoring > ModelArts**.

4. Select a real-time service for which you want to create an alarm rule and click **Create Alarm Rule** in the **Operation** column.

5. On the **Create Alarm Rule** page, create an alarm rule for ModelArts real-time services and models as prompted.

Method 3: Setting an Alarm Rule for a Model Version

1. Log in to the management console.

2. On the **Service List**, click **Cloud Eye** under **Management & Governance**.

3. In the left navigation pane, choose **Cloud Service Monitoring > ModelArts**.

4. Click the down arrow next to the target real-time service name. Then, click **Create Alarm Rule** in the **Operation** column of the target version.

5. On the **Create Alarm Rule** page, create an alarm rule for model loads as prompted.

Method 4: Setting an Alarm Rule for a Metric of a Service or Model Version

1. Log in to the management console.

2. On the **Service List**, click **Cloud Eye** under **Management & Governance**.

3. In the left navigation pane, choose **Cloud Service Monitoring > ModelArts**.

4. Click the down arrow next to the target real-time service name. Then, click the target version and view alarm rule details.

5. On the alarm rule details page, click the plus sign (+) in the upper right corner of a metric and set an alarm rule for the metric.

5.3 Viewing Monitoring Metrics

Scenario

Cloud Eye on the cloud service platform monitors the status of ModelArts real-time services and model loads. You can obtain the monitoring metrics of each ModelArts real-time service and model loads on the management console. Monitored data requires a period of time for transmission and display. The status of ModelArts displayed on the Cloud Eye console is usually the status obtained 5 to 10 minutes before. You can view the monitored data of a newly created real-time service 5 to 10 minutes later.

Prerequisites

- The ModelArts real-time service is running properly.
- Alarm rules have been configured on the Cloud Eye page. For details, see **Setting Alarm Rules**.
- The real-time service has been properly running for at least 10 minutes.
- The monitored data and graphics are available for a new real-time service after the service runs for at least 10 minutes.
- Cloud Eye does not display the metrics of a faulty or deleted real-time service. The monitoring metrics can be viewed after the real-time service starts or recovers.

Monitoring data is unavailable without alarm rules configured on Cloud Eye. For details, see **Setting Alarm Rules**.

Procedure

1. Log in to the management console.
2. In the **Service List**, click **Cloud Eye** under **Management & Governance**.
3. In the left navigation pane, choose **Cloud Service Monitoring > ModelArts**.
4. View monitoring graphs.
   - Viewing monitoring graphs of a real-time service: Click **View Metric** in the **Operation** column.
   - Viewing monitoring graphs of the model loads: Click ⌄ next to the target real-time service, and click **View Metric** in the **Operation** column of the target model.
5. In the monitoring area, you can select a duration to view the monitoring data.

   You can view the monitoring data in the recent 1 hour, 3 hours, or 12 hours.

   To view the monitoring curve of a longer time range, click ⤢ to enlarge the graph.