ModelArts

Inference deployment

Issue	01
Date	2026-07-07

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

Notice

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

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

Huawei Cloud Computing Technologies Co., Ltd.

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

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

Contents

1 Inference Deployment (New Version)

1.1 Deploying and Using Real-Time Inference

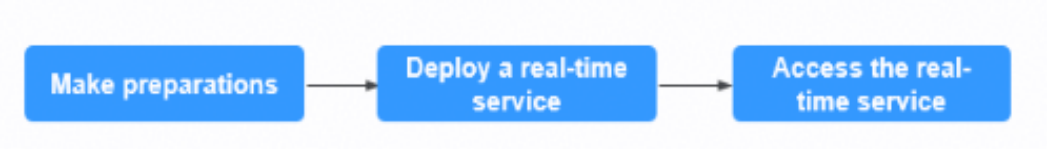
NOTE

The one-click deployment feature for real-time services is only available in the new console in the CN Southwest-Guiyang1 region.

ModelArts helps you quickly deploy AI models as ready-to-use inference services and offers APIs to easily incorporate these services into your custom applications.

ModelArts allows you to deploy a model as a real-time service that provides a real-time test UI and monitoring capabilities. This service provides a callable API. Real-time inference is used in situations that need fast responses, like online intelligent customer service and autonomous driving decisions.

Figure 1-1 Real-time inference deployment process



Preparations

- Create a dedicated resource pool and ensure that the target resource pool has sufficient resources for deployment. For details, see **Creating a Standard Dedicated Resource Pool**.
- Prepare the image beforehand. For details, see **Applications of Custom Images**.
- Prepare the model and code file and upload them to an **OBS bucket**, **OBS parallel file system**, or **SFS Turbo file system**.

Deploying a Real-Time Service

ModelArts enables cloud deployment, which involves running inference services using cloud infrastructure like servers, storage, and networks. This approach works best for tasks needing significant compute and handling large datasets.

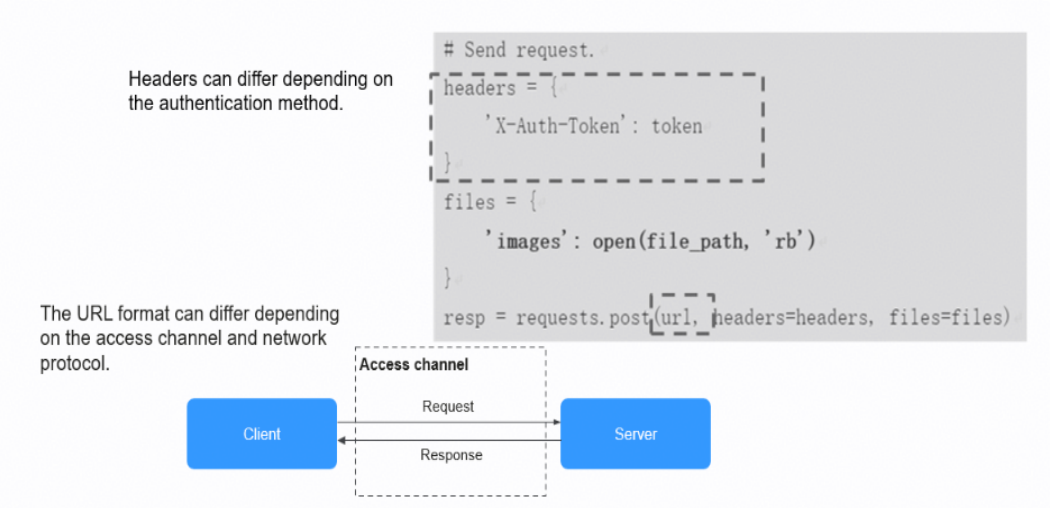
Cloud-based deployment uses real-time inference to process individual requests instantly and deliver results immediately. ModelArts allows you deploy models into web services with internet-accessible APIs. It includes a UI for testing and monitoring these services. Once deployed, the service offers RESTful APIs for sending requests and receiving responses. Real-time inference is used in situations that need fast responses, like online intelligent customer service and autonomous driving decisions.

ModelArts allows you to deploy models as real-time inference services. For details, see **Deploying a Real-Time Inference Service Using a Single Node**.

Accessing a Real-Time Service

If a real-time service is in the **Running** state, it has been deployed. This service provides a standard RESTful API for calling. When accessing a real-time service, you can choose the authentication method, access channel, and transmission protocol that best suit your needs. These three elements make up your access requests and can be mixed and matched without any interference. For example, you can use different authentication methods for different access channels and transmission protocols.

Figure 1-2 Authentication method, access channel, and transmission protocol



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

- **No authentication**: No authentication is required.
- **Token-based authentication**: Use Huawei Cloud Identity and Access Management (IAM) for authentication. The validity period of a token is 24 hours. When using a token for authentication, cache it to prevent frequent calls.

- **API key authentication**: API key authentication provides a straightforward method for securing APIs with basic access control needs. Create an API key in the Huawei Cloud console and include it in the request header for API calls.

ModelArts allows you to call APIs to access real-time services in the following ways (HTTPS requests are used as examples):

- **Accessing a Real-Time Service Through a Public Network**: By default, ModelArts inference uses the public network to access real-time services. A standard, callable RESTful API is provided after deployment of a real-time service.
- **Accessing a Real-Time Service Through a Private Network**: ModelArts offers private network connection. When you create a private network connection request, it automatically sets up a VPCEP to connect your VPC with the real-time inference service securely.

Real-time service APIs are accessed using HTTPS by default. Additionally, the following transmission protocols are also supported:

- **Accessing a Real-Time Service Using WebSocket**: 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.
- **Accessing a Real-Time Service Using Server-Sent Events**: Server-Sent Events (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.

1.2 Performing Custom Service Deployment

1.2.1 Configuring Service Information

NOTE

Real-time services have two versions. The new version is recommended.

Overview

The inference service information provides a central portal to ensure a permanent and stable call URL even as your models iterate. You can easily define authentication methods, network access policies, and global traffic switches here.

Billing

Service information is not billed.

Procedure

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.

2. In the real-time service list, click **Deploy**.
3. On the displayed page, configure service information.

Figure 1-3 Basic information

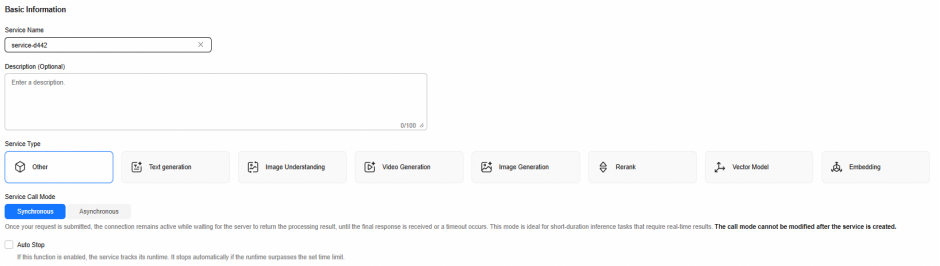


Table 1-1 Basic information

Parameter	Description	Example
Service Name	Name used to identify and manage the real-time service. Enter a name as prompted. Only letters, digits, hyphens (-), and underscores (_) are supported. Max length: 128 characters	**service-e16d** The name is automatically generated by the system.
Description (Optional)	Brief description for a real-time service.	/
Service Type	Real-time service type. You can evaluate the performance of your text generation service after deployment. Supported types include text generation, image understanding, video generation, image generation, reranking, vector, embedding, and other.	**Text generation**

Parameter	Description	Example
Service Call Mode	Real-time services support synchronous and asynchronous calls. • Synchronous call: Once your request is submitted, the connection remains active while waiting for the server to return the processing result, until the final response is received or a timeout occurs. This mode is ideal for short-duration inference tasks that require real-time results. The call mode cannot be modified after the service is created. • Asynchronous call: Once your request is submitted, the service will immediately acknowledge receipt and process it on the backend. You can retrieve inference results later via polling. This mode is ideal for scenarios with long inference times, such as AIGC or video processing, where long-connection timeouts may cause request failures or load imbalances across instances. The call mode cannot be modified after the service is created. For details, see **Asynchronous Inference**.	**Synchronous**
Max Tasks per Service	When the service call mode is set to asynchronous, you need to set the maximum number of tasks that can be created for a single service. The value ranges from 0 to 10,000. Tasks in the **succeeded**, **failed**, **created_failed**, **start_failed**, **upgrade_failed**, **timeout**, or **deleted** state do not occupy the task quota. You can obtain the task status by calling the API for **viewing task details**.	/
Auto Stop	Auto-stop timer. Default: 1 hour; maximum: 24 hours. When auto stop is enabled, the system tracks how long the service runs. It will shut down the service if the runtime goes beyond the set limit. After the real-time service is deployed, you can choose **Model Inference** > **Real-Time Inference** on the console. Choose **More** > **Configure Auto Stop** and reconfigure the auto stop time.	Select it.

Figure 1-4 Network settings

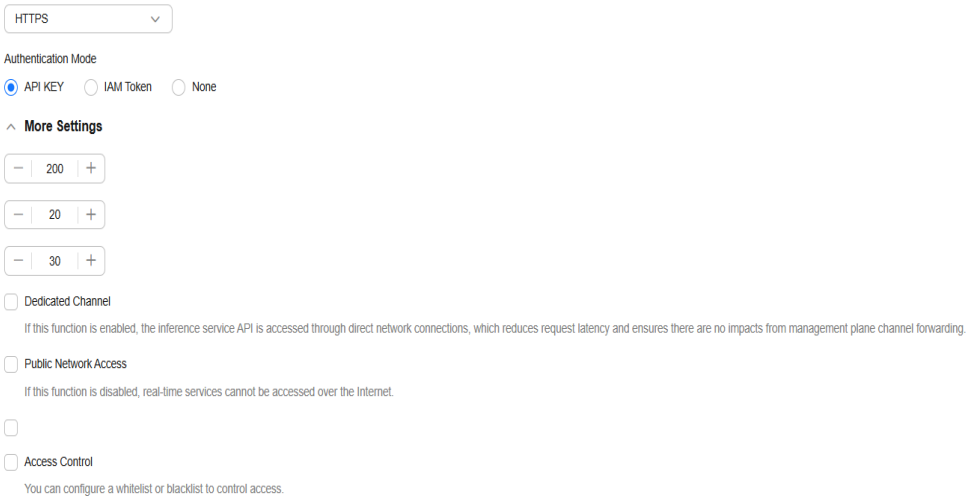


Table 1-2 Network settings

Parameter	Description	Example
Service Access Method	Default: Leverages ModelArts capabilities to provide limited public network access via the platform's default prediction URL. Supports authentication on the platform.	Retain the default settings.
Service Protocol	Protocol used by the inference API and model process API provided by the real-time service. Set this parameter based on the defined inference API. Available protocols: HTTPS, HTTP, WSS, and WS. Using HTTP or WS plaintext can leave you vulnerable to threats like data leakage, tampering, traffic hijacking, and phishing attacks. Proceed with caution.	**HTTPS**
Authentication Mode	**API KEY**, **IAM Token**, and **None** are supported. To use an API key for authentication, create one and bind it to the service on the **API Key Authorization Management** page after deploying the real-time service. Selecting **None** allows the client to call the API directly, which compromises security. This option is only suitable for temporary testing or internal network calls. Avoid using this mode on external networks. For details, see **Accessing a Real-time Service via Different Authentication Methods**.	**API KEY**

Parameter	Description	Example
More Settings	• **Public Network Access**: Specifies whether to allow external network access to the real-time service. This function is enabled by default. If this function is enabled, external networks can access the service. View the API URL on the service details page. If this function is disabled, real-time services cannot be accessed over the Internet. • **Auto-approved Private Network Connection**: Specifies whether private network connection to real-time services requires approval. When this function is enabled, private network connection requests from third-party users will be approved automatically. If this function is disabled, approval is required. For details about how to access a service through a private network, see **Accessing a Real-Time Service Through a Private Network**. • **Access Control**: Once selected, you can specify a whitelist or blacklist for access control. Whitelist: Only user IP addresses from the CIDR blocks configured here are allowed access. You can add up to ten regular expressions. Blacklist: Only user IP addresses from the CIDR blocks configured here not allowed access. You can add up to ten regular expressions.	Retain the default settings.

Figure 1-5 High availability settings

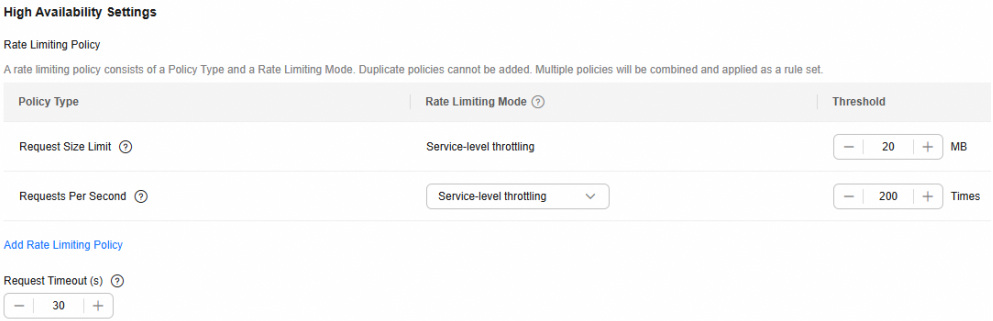


Table 1-3 High availability settings

Parameter	Description	Example
Rate Limiting Policy	A rate limiting policy consists of a **Policy Type** and a **Rate Limiting Mode**. Duplicate policies cannot be added. Multiple policies will be combined and applied as a rule set. For details, see **Table 1-4**.	Retain the default settings.
Request Timeout (s)	The maximum time to wait for the system to return the first token result after a request is sent. The value must range from 1 to 1,200. If no response is received within this period, the request is automatically terminated. In streaming transmission scenarios, the timeout interval is refreshed each time a response to a request is received. However, the end-to-end response timeout interval of the system is fixed at 3,600 seconds.	Retain the default settings.

Table 1-4 Rate limiting policy parameters

Parameter	Sub-Parameter	Description	Example
Policy Type	Request Size Limit	The maximum size allowed for data contained within a single request. Request size range: 1 to 50. The unit is MB.	Retain the default settings.
	Requests Per Second	The maximum number of times the service can be accessed per second. Unit: times. The value must range from 1 to 10,000.	Retain the default settings.

Parameter	Sub-Parameter	Description	Example
Rate Limiting Mode	Service-level throttling	Sets a fixed total rate limit for the service. New requests are rejected immediately once the threshold is exceeded.	Retain the default settings.
	Instance-level throttling	The actual service rate is dynamically adjusted based on the number of deployed replicas. Actual service rate = Deployment replica threshold x Number of running deployment replicas.	Retain the default settings.

Figure 1-6 Advanced settings

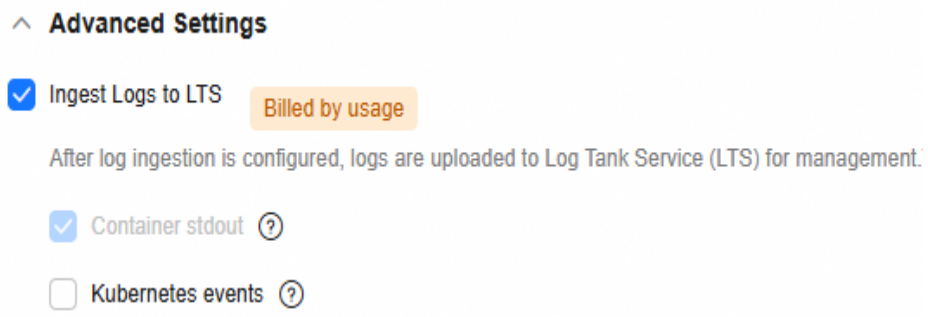


Table 1-5 Advanced settings

Parameter	Description
Ingest Logs to LTS	If **Ingest Logs to LTS** is not enabled on the service deployment page, only real-time service logs are displayed. When this function is enabled, logs are uploaded to Log Tank Service (LTS) for management. You are billed on a pay-per-use basis. For details, see **LTS Pricing Details**. LTS automatically creates log groups and streams and caches logs for seven days by default. You can search for and analyze runtime logs on LTS. • Log search: Search logs using specific keywords or phrases. Narrow your results by selecting a specific time range to find events and issues during that period. • Statistical charts: After sending logs to LTS, use SQL analysis syntax to find important log data and view the results as statistical charts. • Log analysis: Before searching for analyzing logs, set up structured data and indexing for them. • Real-time logs: Once you connect your real-time service logs to LTS, they will be sent every minute. You can view these updates from the **Real-Time Logs** tab, where you can also easily search and analyze the data. For details, see **Log Tank Service**.
Container stdout	After you select **Ingest Logs to LTS**, **Container stdout** is selected by default and cannot be modified. Collect all container standard output and report it to Log Tank Service (LTS). The log retention period follows the setting of the corresponding log group, which is 30 days by default: • For dedicated resource pools, the LTS log groups and log streams are those created and selected during the installation of the resource pool log collection plugin. • For public resource pools, the system automatically creates LTS log groups and log streams. The naming conventions are: Log group: Modelarts-Infer-Log-Group-{NUM}; log stream: Inf-Stdout-{serviceName}-{First 8 characters of converted serviceId}.

Parameter	Description
Kubernetes events	After you select **Ingest Logs to LTS**, you can manually select **Kubernetes events**. Once enabled, Kubernetes events (pod events) will be collected and reported to LTS. By default, logs are retained for 7 days. You can view pod events in the event list on the inference service details page. For details, see **Viewing Events of a Real-Time Service**. If this option is not selected, you can only view pod events of the last hour in the event list on the inference service details page.
Tags	ModelArts can work with Tag Management Service (TMS). When creating resource-consuming tasks in ModelArts, for example, training jobs, add tags for these tasks so that ModelArts can use tags to manage resources by group. 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. Custom tags are available only to the service resources of the user who has created the tags. For details about how to use tags, see **Using TMS Tags to Manage Resources by Group**.

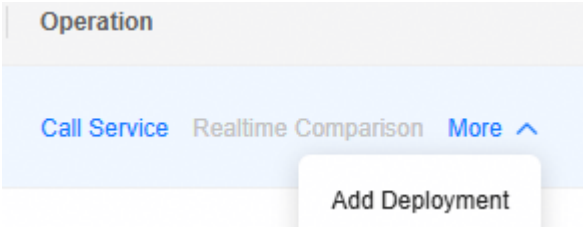
4. After configuring the service information, click **Create Service Only**. The system automatically returns to the real-time inference service list page.

   You can also click **Next** to continue the service deployment configuration.

Follow-Up Operations

In the real-time inference service list, choose **More** > **Add Deployment** in the **Operation** column and configure the inference deployment information. For details, see **Deploying a Real-Time Inference Service Using a Single Node**.

Figure 1-7 Add Deployment



1.2.2 Deploying a Real-Time Inference Service Using a Single Node

NOTE

Real-time services have two versions. The new version is recommended.

Overview

In a single-node deployment, one inference unit completes all tasks. It handles all computing, pre-processing, and post-processing on the same instance. This setup is lightweight, simple, and easy to maintain. Only one inference unit is needed to provide the service.

Use cases:

- Deployment of small-to-medium-sized models with fewer parameters (e.g., classification, detection, and lightweight dialogue models).
- Low-concurrency, low-traffic businesses (low daily request volume and low peak traffic).
- Development, testing, validation environments, and Proof of Concept (POC) tests.
- Budget-constrained scenarios prioritizing low costs without the need for large-scale scaling.
- Scenarios requiring rapid launch, quick validation, and simplified O&M.

Deployment mode comparison:

- Single-node deployment (basic mode): A single inference unit is sufficient to complete inference tasks. It features a simple architecture, eliminates the need for cross-node communication, and incurs low deployment and O&M costs. However, due to the hardware resource constraints of a single machine, it cannot host ultra-large parameter models, and its concurrency and throughput capabilities are limited. The processing capacity of the model service can be enhanced by increasing the number of deployment replicas. **This topic focuses primarily on single-node deployment.**
- **Multi-node prefill-decode (PD) co-location (multi-role separation – co-location)**: PD co-location refers to deploying both the prefill and decode stages of LLM inference onto the same set of compute nodes (such as NPUs/GPUs) to share KV cache resources. This mode is suitable for resource-constrained scenarios or when architecture simplification is required. When using the vLLM framework in a co-location scenario, you are advised to set the first unit as the vLLM master node and place the remaining worker nodes in other units. For details about PD co-location, see **Deploying a Real-Time Inference Service Using Multi-Node PD Co-location**.
- Multi-node PD disaggregation (multi-role separation – isolation): In the multi-role separation mode, the prefill and decode units are deployed on different physical nodes to achieve complete resource isolation. This mode delivers optimal performance but incurs higher costs. For details about multi-node PD disaggregation, see **Deploying a Real-Time Inference Service Using Multi-Node PD Disaggregation**.

Constraints

- Deployment limit: A user can create up to 20 real-time services.
- Deployment mode: Only the basic mode (non-multi-role separation/PD disaggregation) is supported.
- Node limit: Single-node deployment is required. Cross-node deployment and distributed scaling are not supported.
- Performance upper limit: The GPU, CPU, and memory of a single node are limited. High concurrency may cause out-of-memory (OOM) and increase latency.
- Low availability: If a node is faulty, the service is interrupted, and no automatic switchover is available.
- O&M restrictions: P/D ratio optimization and intelligent O&M are not supported.
- Scaling constraints: Only horizontal scaling by adding replicas (on the same node or nodes of the same specifications) is supported; role separation is not permitted.

Prerequisites

- You have prepared data as instructed in **Preparations**.
- Your account is not in arrears to ensure available resources for running services.
- You have configured the inference service information as instructed in **Configuring Service Information**.

Notes

Resource pools allocate quota for real-time services even when they are **Abnormal** or **Stopped**. If the quota is insufficient and no more services can be deployed, delete some abnormal services to release resources.

- Quota calculation:

  The quota stays the same when you deploy real-time services with a dedicated resource pool. It only changes if you create, modify, or delete a resource pool.
- Usage metering:

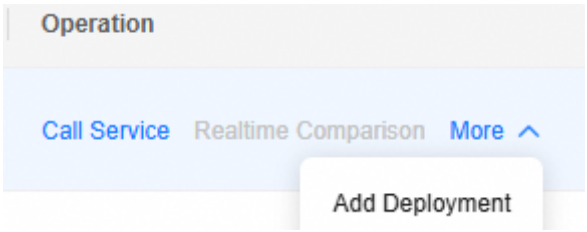
  Deploying real-time services in a dedicated resource pool is not metered. Only the usage of the dedicated resource pool itself is metered.
- Mounting SFS Turbo:

  Before mounting an SFS Turbo file system to a real-time service, **associate the dedicated resource pool network with the file system**.

Configuring Deployment Settings

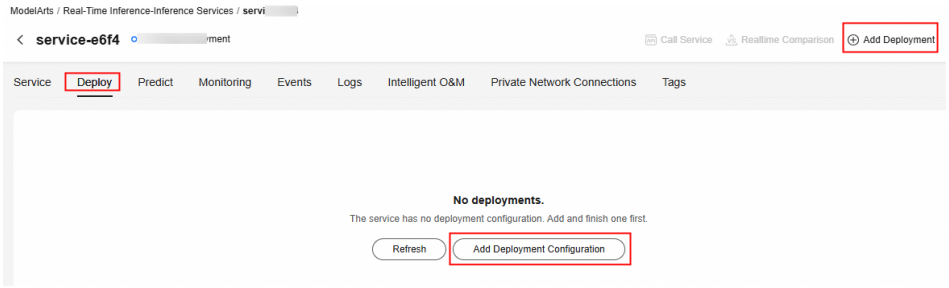
1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. On the service list page, choose **More** > **Add Deployment** on the right of the service name.

Figure 1-8 Adding a deployment



You can also click the service name to go to the service details page and configure deployment information on the **Deploy** tab page or by clicking **Add Deployment** in the upper right corner of the service details page.

Figure 1-9 Adding a deployment



2. On the **Deploy Real-Time Service** page, set basic information, resource settings, model settings, unit settings, deployment management settings, and advanced settings, and click **Confirm Deployment**. The **Confirmation** page is displayed, as shown in **Confirming Settings**.

- Basic information

Table 1-6 Basic information

Parameter	Description	Example
Deployment Name	Name of the current deployment, which is used to identify and manage the deployment configuration of the real-time service. Enter a name as prompted. Only letters, digits, hyphens (-), and underscores (_) are supported. Max length: 128 characters	**service**
Description (Optional)	Brief description of the deployment.	/

- Resource settings

Table 1-7 Resource settings

Parameter	Description	Example
Resource Pool Type	• **Public Resource Pool** Public resource pool for deploying the real-time service. The public resource pool supplies shared compute clusters assigned according to job parameters. Each job operates with its own isolated resources. This option offers cost-effective and flexible solutions for tasks like development and testing. Choosing the public resource pool might leave fewer resources available because of its limits. If this happens, join the queue and wait your turn. • **Dedicated Resource Pool** Dedicated resource pool for deploying the real-time service. The resources provided in a dedicated resource pool are exclusive and more controllable. Use dedicated resource pools for core production services to secure exclusive resources.	**Dedicated Resource Pool**
Resource Pool	If **Resource Pool Type** is set to **Dedicated Resource Pool**, click **Select Resource Pool**, select the corresponding specifications in the dedicated resource pool specifications area, and click **OK**. The physical pools with logical subpools created are not supported. If no dedicated resource pool is available, create one. You can choose a heterogeneous resource pool for deploying real-time services. If you use a heterogeneous dedicated resource pool, ensure the real-time service's instance specifications match the pool's specifications.	/
Deployment Replicas	A deployment instance consists of units capable of independently completing an inference task. Specify the number of instance replicas for the deployment. The value ranges from **1** to **128**. If there is one replica, only one service instance runs. This is a standard single-node setup. With three replicas, three identical service instances run simultaneously. This spreads out requests, improves handling multiple tasks, and provides basic high availability. In single-node setups, the number of replicas depends on the node's memory and hardware. Too many replicas can cause memory shortages, OOM errors, compute bottlenecks, and service freezes. We recommend deploying 1 to 4 replicas in single-node setups.	**1**

Parameter	Description	Example
Scheduling Policy	Two scheduling policies are available: **HA scheduling** and **Compact scheduling**. **HA scheduling** is enabled by default. • **HA scheduling**: Pods from different replicas will be distributed across different nodes as evenly as possible, while multiple pods under the same replica will prioritize scheduling to the same node to ensure high availability for inference services. If both **HA scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence. • **Compact scheduling**: Enable bin packing for cluster workloads. The scheduler will prioritize placing pods on nodes with higher resource consumption to reduce idle resource fragmentation and improve cluster resource utilization. When both **Compact scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence. Affinity scheduling is set in **More Settings** of the **Unit Settings**. For details, see **Table 1-9**.	**HA scheduling**
Scheduling Priority	Resource scheduling priority for service deployment. When **Service Resource Pool Settings** is set to **Dedicated resource pool**, set this parameter. The platform handles jobs by prioritizing them from highest to lowest. Range: 1 (lowest) to 3 (highest). If multiple jobs have the same priority, they are scheduled in the order they were submitted. The final schedule depends on available resources. When resources are enough and priorities match, the first jobs submitted are scheduled first.	**1**

- Model settings

Table 1-8 Model source parameters

Category	Parameter	Description	Example
Model Source	Platform asset	When deploying a real-time service, you can select either **Platform asset** or **Custom Model** as the model source. When selecting **Platform asset**, the model originates from the ModelArts asset center. Click a card to select a model. You can choose from either **Preset Models** or **My Models**: • **Preset Models**: These are inference model assets built into ModelArts that you can select and use directly. • **My Models**: These are models generated by importing a local model or by completing model pre-training or model fine-tuning on ModelArts. Select your model and click **OK**. Supported model assets depend on the resource pool specifications. Select a dedicated resource pool or switch to the public resource pool and try again.	**Platform asset**
	Custom Model	When selecting a custom model, you can select the model storage type. Set the model storage address and mount path. Supported storage types for custom models: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. For details about the parameters, see **Storage Mounting**.	**Custom Model**

- Unit settings

Table 1-9 Inference unit parameters

Parameter	Description	Example
Basic mode	Inference service deployment supports **Basic mode** and **Multi-role**. **Basic mode**: Requires configuring only one inference unit, which independently hosts the entire inference service. Select **Basic mode** for single-node deployment. For details about multi-role separation deployment, see **Deploying a Real-Time Inference Service Using Multi-Node PD Co-location**.	**Basic mode**

Parameter	Description	Example
Unit Replicas	If the deployment mode is **Basic mode**, the number of unit replicas is **1** by default and cannot be changed.	**1**
Unit Name	Enter a custom unit name ranging from 0 to 16 characters. Only lowercase letters, digits, and hyphens (-) are supported, and it must start and end with a letter or digit.	**role-0**
Specification Type	Select the hardware resource type when using a dedicated resource pool. Public resource pools only support preset specifications.	**Preset**
Specification Type > Preset	Choose a specification and number of instances per replica. Total resource requirements of the inference unit = Unit instance specifications x Number of resource instances. For example, if the unit instance specification provides eight accelerators and you need 32 in total, enter **4**.	/

Parameter	Description	Example
Specification Type > Custom	Use a custom specification if the preset ones do not fit your needs. GPU virtualization is enabled for this resource pool. Resources will be strictly allocated based on request volume. Configure specific vCPUs and memory. For details, see **Viewing Details About a Real-Time Service**. • CPU cores: The value must be at least 0.01 with exactly two decimal places and should not exceed the total cores available in the resource pool. • Memory: The memory value must be an integer of at least 4 and should not exceed the total memory available in the resource pool. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead. When deploying a real-time service, you may select a heterogeneous dedicated resource pool—one that contains multiple architectures or specifications. For example, a resource pool might include node pool 1 (x86 architecture, CPU-only, 8 vCPUs, 32 GB), node pool 2 (x86 architecture, CPU-only, 8 vCPUs, 64 GB), and node pool 3 (Arm architecture, NPU-accelerated, 192 vCPUs, 1536 GB). When deploying a real-time service using such a heterogeneous resource pool, you can first select the node pool specifications, for example, node pool 1 with CPU-only specifications, and then select the job specifications required for deploying the service, such as the number of CPUs, memory, or accelerators required for deploying the real-time service. If you have node pools with the same specification, like node pool 1 and node pool 2, and you choose node pool 1 during setup, the service may still be deployed in either node pool 1 or node pool 2. This depends on which pool has better resource usage at the time. When deploying multiple inference units, you cannot use heterogeneous node pools. All inference units must be in the same node pool.	/

Parameter	Description	Example
Image Type	Select the source of the inference image. • Preset image: Use images from ModelArts asset management. You can tag them with details like supported specifications and frameworks for easy management. For details, see **ModelArts Images**. • Custom image: Select your custom image. For details about how to create a custom image, see **Creating a Custom Image for a Model**. Use your own image via any of the following options: – **SoftWare Repository for Container (SWR)**: SWR is a secure, reliable, and easy-to-use container image management service that supports full lifecycle management of container images. – Registered image: Select an image registered with ModelArts. • Image URL: Enter the image path of a custom image. • **Pre-warmed image**: Select an image pre-warmed in dedicated resource pool in advance. If the image warmup task is done and the status is normal, deployment speeds up automatically. The selected image path and the service to be deployed must be in the same region.	**Custom Images**

Parameter	Description	Example
Environment Variables (Optional)	Inject environment variables into the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables. Environment variables are key-value pairs. For example, the key is **A** and the value is **AAAA**. Only letters, digits, underscores (_), hyphens (-), and periods (.) are supported. They cannot start with a digit and cannot exceed 64 characters. The value of an environment variable cannot be in HTML format, such as <p>, <^>, and <...>. You can upload an Excel file to batch import environment variables. Only .xlsx and .xls files are supported, and up to 100 parameters can be uploaded. To ensure correct parsing, fill in the file strictly according to the template format. To download the template, click **Download Template**. Click **Local Upload**. In the dialog box that is displayed, click **Add File** to import the local Excel file. Check the parsed environment variable key values. If the check result is **To be modified**, modify the values as required and upload the file again. After the check result is **Passed**, select the key values and click **OK** to complete the batch upload of environment variables.	/
Mount File Storage	Mount persistent storage for data access or artifact dumping. • Supported storage types for file storage: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. • Supported storage type for artifact dumping: OBS parallel file system. You can add up to 15 storage paths for file storage and 10 storage paths for artifact dump. Available storage types are subject to the GUI. Artifact dump cannot share the same OBS bucket or PFS with file or model mounting. For details about storage parameters and requirements, see **Storage Mounting**.	**Object Storage Service – Bucket**

Parameter	Description	Example
Health Check	Configure probes to monitor the model health. It can only be configured when the health check API is configured in the custom image. Otherwise, the model deployment fails. The following probes are supported: • **Startup Probe**: This probe checks if the instance has started. If a startup probe is configured, the liveness or readiness probe is not executed until the startup probe is successful, allowing sufficient time for the application to complete initialization. If the startup probe fails, the instance is restarted. If startup probe is not configured, the service status changes to success immediately after the service is scheduled. The service may be in the **Running** state and the prediction cannot be performed because the model is being loaded. • **Readiness Probe**: This probe verifies whether the instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. For more information about probe configurations and descriptions, see **Real-Time Service Health Check**.	/
Boot Command	Set the service boot command.	/
More Settings > Automatic Rebuild	When enabled, if a pod restarts due to deployment configuration changes or failures, the platform will automatically rebuild it using the selected policy. If disabled, the platform will not intervene. For details, see **Auto Rebuild upon a Real-Time Service Fault**.	/
More Settings > Auto Restart	When enabled, if an NPU, switch, or hardware fault is detected, services on the faulty node are automatically rescheduled. Some capabilities are only supported by Snt9b and Snt9b2 resources. For details, see **Auto Restart upon a Real-Time Service Fault**. To ensure service continuity, configure multiple instances.	/

Parameter	Description	Example
More Settings > Graceful Shutdown	Enabling this feature allows you to configure shutdown timeouts and commands. This prevents in-progress requests from being forcefully interrupted, thereby enhancing the availability and stability of the system. If you have configured health checks and set a large sleep value in this command, it will result in a longer restart or stop time for the container when the health check fails. • **Shutdown Timeout (s)**: This parameter indicates the maximum length of time that can pass between when a Pod receives a stop signal to when it is forcefully stopped. It is used for the Pod to perform cleanup operations (such as closing connections, releasing resources, and saving states). • **Shutdown Command**: The shutdown command is triggered when the container receives a stop signal, but it must be completed within the grace period or the shutdown times out. If this happens, the container will be forcefully stopped. You can use this command to perform operations such as closing database connections, releasing file handles, and stopping child processes.	/

Parameter	Description	Example
More Settings > Affinity Scheduling	You can configure the node affinity type and strength to flexibly schedule workloads in a resource pool. If no nodes are specified, the pods will be randomly scheduled according to the default cluster scheduling policy. After this function is enabled, you can refine the pod deployment policy. • Node affinity (strong): The pod must be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node affinity (weak): The system will try to place the pod on the specified node, but it is not guaranteed. • Node anti-affinity (strong): The pod must not be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node anti-affinity (weak): The system will try to avoid deploying the pod on the specified node, but it is not guaranteed In the **Add Node** list, select the nodes that meet the preceding configuration rules. When you choose a pre-warmed model, only the pre-warmed nodes appear on the affinity scheduling page. Unwarmed nodes do not show. A message appears stating: The selected model is pre-warmed and will deploy automatically on the best node. Specifying a node might cause warmup to fail.	/
More Settings > Container User ID	If this option is selected, enter the user ID and user group ID (optional).	/

Parameter	Description	Example
Authentication Credential	Displays when **System Log Reporting** is selected in **Advanced Settings**. A secret is used to verify identity and authorize access. In the information security and identity authentication fields, a secret is a key mechanism to ensure that only authorized users can access the system, resources, or services. • If the version of CCE Container Storage (Everest) in your dedicated resource pool is v2.4.204 or later and the cluster version is v1.28 or later, temporary credentials are enabled by default (no AK/SK is required, ensuring higher security). • If your CCE Container Storage (Everest) version is too low or the cluster does not support temporary credentials, you must mount the credentials using DEW. When using DEW secrets to mount storage, you must include 'accessKeyId' and 'secretAccessKey' (corresponding to your AK and SK, respectively). Ensure the information provided is correct, or the function may not work as expected. To create a secret, click **Create Secret** to go to the DEW console. For details, see **Creating a Secret**. Set the AK/SK in the **Secret key/value** tab. Enter **accessKeyId** and **secretAccessKey** in the **Key** column. To obtain the values, go to the DEW console, and choose **My Credentials** > **Access Keys**.	/

- Deployment management settings

Table 1-10 Deployment management settings

Parameter	Description
Container Protocol	The network transmission protocol used by the container. Configure it according to the API definitions in your actual setup. If **Service Protocol** is set to **HTTP** or **HTTPS** in **Table 1-2**, **Container Protocol** can be set to **HTTP** or **HTTPS**. If **Service Protocol** is set to **WSS** or **WS** in **Table 1-2**, the container protocol is the same as the service protocol by default and is not displayed on the console.
Container Port	The port number that the image listens on. Requests are sent to the instance through this port.

Parameter	Description
More Settings > Deployment Timeout (Minutes)	The wait time before a single service times out. This value includes both the deployment and startup time. Set this parameter properly. The system will cancel the deployment if it exceeds the time limit.
More Settings > Max Surge Replicas (%)	The maximum number of replicas that can be created above the target count during a rolling upgrade When percentages are used, the instance count is rounded up. Example: When **Max Surge Replicas (%)** is set to **1%** during a rolling upgrade of four instances, you can add one new instance at a time. When **Max Surge Replicas (%)** is set to **100%** during a rolling upgrade of two instances, you can add two new instances immediately.

Parameter	Description
More Settings > Max Unavailable Replicas (%)	The maximum number of replicas that can be unavailable relative to the target count during a rolling update When percentages are used, the instance count is rounded down. Example: ● When **Max Unavailable Replicas (%)** is set to **1** during a rolling upgrade of four instances, four instances must remain available. An old instance can only be deleted after its replacement starts running. ● When **Max Unavailable Replicas (%)** is set to **50** during a rolling upgrade of two instances, one old instance can be deleted immediately to release resources, but at least one instance must remain available. ● If you deploy a service with **Max Unavailable Replicas (%)** at **20**, requiring five instances, and only one fails due to lack of resources while four start successfully, the deployment succeeds, and the service status becomes **Running**. If **Max Unavailable Replicas** equals the target replica count, there is a risk of downtime (**Min Available Replicas** = **Total Replicas** – **Max Unavailable Replicas**). Example: Setting **Max. Invalid Instances** to **100** allows the service stay running when an instance fails because of quick recovery. Yet, the service will not be able to perform predictions, potentially leading to interruptions. To ensure a hitless upgrade, set Max. Invalid Instances to 1%. (This requires extra resources. Make sure you have enough, or the update will fail.)

- Advanced settings

Table 1-11 Advanced settings

Parameter	Description
Key Settings	Inference services support secret mounting. This uses encryption to protect sensitive data and securely mount it to containers. To protect your data, do not enter sensitive information in plaintext. • **Custom key**: Enter the key, key value, and mount path. • **DEW**: Data Encryption Workshop (DEW) is a comprehensive cloud data encryption service, including Key Management Service (KMS), Key Pair Service (KPS), and Dedicated Hardware Security Module (Dedicated HSM). To use DEW to configure the key, go to the DEW console to **create a key**, select the DEW key on the ModelArts console, and enter the mount path. Choose whether to select **Associate Image User Group ID**. The user group ID of the user who starts the image can be associated. After the association, the secret mounting security is improved. For details, see **Secret Mounting**.

Parameter	Description
Intelligent Routing Policy	If intelligent routing is enabled, set intelligent routing policies. The following policies are supported: • Round robin: Tasks are distributed sequentially to different nodes in order. This ensures an even distribution across the cluster and achieves load balancing. • Source IP hash: The system calculates hash values using client IP addresses to route requests from the same IP address to the same node. • Least connections: Requests are routed to the node with the fewest real-time connections. This method ensures that the load is distributed more evenly, improving service stability and response time. • Minimum time to first token: Requests are routed to the node with the lowest average time to first token. This method aims to minimize the wait time between receiving a request and starting its processing. This latency applies even when system resources are available immediately. • Overall loads: Requests are routed to the node with the lowest overall pressure, considering factors such as the number of connections, time to first token, and custom metrics. It directs new requests to less busy nodes to avoid resource waste and overloading. • SLO priority: Services are prioritized according to their assigned Service Level Objective (SLO) (0–3, where 0 is the highest). This ensures lower latency for higher-priority workloads. For details, see **Intelligent Routing Policy**.

Parameter	Description
Custom Metric Collection	If this function is enabled, you need to input the metric collection port details. Custom metrics will send data to AOM's Prometheus instances for viewing on the AOM console. To query custom metrics, see **Viewing Performance Metrics of a Real-Time Service on AOM** and **Viewing Performance Metrics of a Real-Time Service on ModelArts**. **Constraints** • ModelArts calls the HTTP API provided in the custom metric configuration every 10 seconds to obtain metric data. • The metric data text returned by the HTTP API provided in the custom metric configuration cannot exceed 32 KB. **Data Format of Custom Metrics** The format of custom metrics data must comply with the open metrics specifications. That is, the format of each metric must be: <*metric_name*>{<*tag_name*>=<*tag_value*>,...} <*sample_value*> [*timestamp_in_millisecond*] The following shows an example (the comment starts with #, which is optional): # HELP http_requests_total The total number of HTTP requests. # TYPE http_requests_total gauge html_http_requests_total{method="post",code="200"} 1656 1686660980680 html_http_requests_total{method="post",code="400"} 2 1686660980681
System Log Reporting	Displays only for NPU dedicated resource pools. When enabled, system logs are mounted to a fixed parallel file system for O&M engineers to analyze. System logs are stored for 30 days and will be automatically deleted after that. The mount path of system logs cannot be changed.

Confirming Settings

On the **Deploy Real-Time Service** > **Confirmation** page, confirm the configuration information and click **Confirm Deployment**.

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.

You can go to the real-time service list to check if the deployment is complete. Once the service status changes from **Deploying** to **Running**, the service is deployed.

NOTE

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

Follow-Up Operations

- For details about how to test a real-time service, compare the model effect in real time, and debug a real-time service using CloudShell, see **Testing a Real-Time Service**.
- For details about how to view real-time service details, see **Viewing Details About a Real-Time Service**.
- For details about how to modify, stop, and delete a real-time service, see **Managing the Lifecycle of a Real-Time Service Deployment**.

1.2.3 Deploying a Real-Time Inference Service Using Multi-Node PD Co-location

NOTE

Real-time services have two versions. The new version is recommended.

Overview

During the large-scale deployment of LLM inference services, the prefill stage is responsible for processing user inputs and generating initial tokens, making it a high-compute, short-duration intensive task. Conversely, the decode stage is responsible for generating text word-by-word and maintaining conversation context, making it a low-compute, long-duration streaming task. Because these two types of tasks have significantly different resource requirements, single-node deployments are prone to resource contention and performance fluctuations. Multi-node PD co-location is suitable for core inference scenarios involving medium-to-large LLMs, moderate concurrency, and cost-prioritized budgets. Specific use cases include:

- Streaming inference services such as LLM dialogue, copywriting generation, and code generation.
- Heterogeneous compute (GPU/NPU) hybrid clusters that need to balance performance and cost.
- Production environments that require a balance between resource utilization and inference stability.
- Medium-scale businesses that do not require complete physical isolation but need to mitigate PD resource contention.
- Unified compute pools for small-to-medium teams featuring multi-model sharing and auto scaling scenarios.

Core Principles

Deployment mode comparison:

- Single-node deployment (basic mode): Features only one inference unit, where prefill and decode run serially within the same container. This results in intense resource contention, high latency fluctuations, and limited throughput. For configuration details, see **Deploying a Real-Time Inference Service Using a Single Node**.
- **Multi-node PD co-location (multi-role separation – co-location)**: PD co-location refers to deploying both the prefill and decode stages of LLM

inference onto the same set of compute nodes (such as NPUs/GPUs) to share KV cache resources. This mode is suitable for resource-constrained scenarios or when architecture simplification is required. When using the vLLM framework in a co-location scenario, you are advised to set the first unit as the vLLM master node and place the remaining worker nodes in other units. **This topic focuses on multi-node PD co-location.**

- Multi-node PD disaggregation (multi-role separation – isolation): In the multi-role separation mode, the prefill and decode units are deployed on different physical nodes to achieve complete resource isolation. This mode delivers optimal performance but incurs higher costs. For details about multi-node PD disaggregation, see **Deploying a Real-Time Inference Service Using Multi-Node PD Disaggregation**.

Core logic of multi-node PD co-location:

- Task splitting: The inference workflow is split into two independent units, prefill and decode, allowing them to scale independently without mutual interference.
- Resource sharing: Both units are deployed in a hybrid fashion within the same pool, with node resources scheduled on demand. Idle nodes are prioritized for high-load tasks, maximizing resource utilization.
- Flexible scheduling: No rigid node affinity is enforced; it supports dynamic cross-node scheduling, making it well-suited for heterogeneous clusters and auto scaling.
- Performance compromise: It mitigates resource contention, delivering lower latency and higher throughput than single-node setups, while offering lower costs and simpler deployment than PD disaggregation.

Constraints

- Deployment mode restrictions: PD co-location is supported exclusively in multi-role separation mode; it is not supported in basic mode.
- Node pool restrictions: Multiple inference units (prefill and decode) must be deployed in the same node pool; cross-heterogeneous node pools are currently not supported.
- Affinity scheduling restrictions: PD co-location does not configure strong affinity or strong anti-affinity; it only supports weak affinity or no affinity configuration.
- Image requirements: The model image must be adapted for multi-role separation architecture and support independent deployment of prefill and decode; otherwise, the deployment will fail.
- Resource specification restrictions: When deploying in heterogeneous resource pools, the instance specifications of the prefill and decode units must match the specifications of the node pool.

Prerequisites

- You have prepared data as instructed in **Preparations**.
- Your account is not in arrears to ensure available resources for running services.
- You have configured the inference service information as instructed in **Configuring Service Information**.

- The current account has the real-time inference deployment permission and dedicated resource pool scheduling permission. For details, see **Configuring Agency Authorization for ModelArts with One Click**.

Notes

Resource pools allocate quota for real-time services even when they are **Abnormal** or **Stopped**. If the quota is insufficient and no more services can be deployed, delete some abnormal services to release resources.

- Quota calculation:

  The quota stays the same when you deploy real-time services with a dedicated resource pool. It only changes if you create, modify, or delete a resource pool.
- Usage metering:

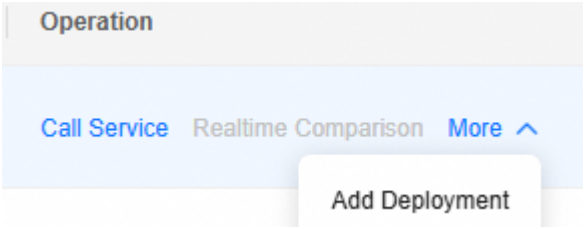
  Deploying real-time services in a dedicated resource pool is not metered. Only the usage of the dedicated resource pool itself is metered.
- Mounting SFS Turbo:

  Before mounting an SFS Turbo file system to a real-time service, **associate the dedicated resource pool network with the file system**.

Configuring Basic Information

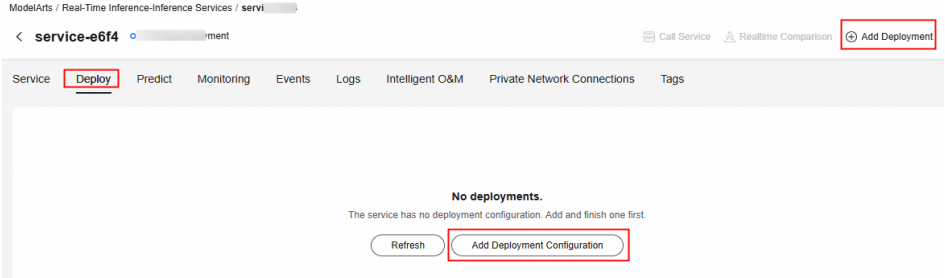
Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. On the service list page, choose **More** > **Add Deployment** on the right of the service name.

Figure 1-10 Adding a deployment



You can also click the service name to go to the service details page and configure deployment information on the **Deploy** tab page or by clicking **Add Deployment** in the upper right corner of the service details page.

Figure 1-11 Adding a deployment



On the **Deploy Real-Time Service** page, configure basic information, resource settings, model settings, unit settings, deployment management settings, and advanced settings.

The following table describes the basic information.

Table 1-12 Basic information

Parameter	Description	Example
Deployment Name	Name of the current deployment, which is used to identify and manage the deployment configuration of the real-time service. Enter a name as prompted. Only letters, digits, hyphens (-), and underscores (_) are supported. Max length: 128 characters	**service**
Description (Optional)	Brief description of the deployment.	/

Configuring Resource Settings

The following table describes the resource settings.

Table 1-13 Resource settings

Parameter	Description	Example
Resource Pool Type	Both dedicated resource pools and public resource pools support multi-role separation deployment. • **Public Resource Pool** Public resource pool for deploying the real-time service. The public resource pool supplies shared compute clusters assigned according to job parameters. Each job operates with its own isolated resources. This option offers cost-effective and flexible solutions for tasks like development and testing. Choosing the public resource pool might leave fewer resources available because of its limits. If this happens, join the queue and wait your turn. • **Dedicated Resource Pool** Dedicated resource pool for deploying the real-time service. The resources provided in a dedicated resource pool are exclusive and more controllable. Use dedicated resource pools for core production services to secure exclusive resources.	**Dedicated Resource Pool**

Parameter	Description	Example
Resource Pool	If **Resource Pool Type** is set to **Dedicated Resource Pool**, click **Select Resource Pool**, select the corresponding specifications in the dedicated resource pool specifications area, and click **OK**. The physical pools with logical subpools created are not supported. If no dedicated resource pool is available, create one. You can choose a heterogeneous resource pool for deploying real-time services. If you use a heterogeneous dedicated resource pool, ensure the real-time service's instance specifications match the pool's specifications.	/
Deployment Replicas	A deployment instance consists of units capable of independently completing an inference task. Specify the number of instance replicas for the deployment. The value ranges from **1** to **128**. If there is one replica, only one service instance runs. This is a standard single-node setup. With three replicas, three identical service instances run simultaneously. This spreads out requests, improves handling multiple tasks, and provides basic high availability.	**1**
Scheduling Policy	Two scheduling policies are available: **HA scheduling** and **Compact scheduling**. **HA scheduling** is enabled by default. • **HA scheduling**: Pods from different replicas will be distributed across different nodes as evenly as possible, while multiple pods under the same replica will prioritize scheduling to the same node to ensure high availability for inference services. If both **HA scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence. • **Compact scheduling**: Enable bin packing for cluster workloads. The scheduler will prioritize placing pods on nodes with higher resource consumption to reduce idle resource fragmentation and improve cluster resource utilization. When both **Compact scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence. Affinity scheduling is set in **More Settings** of the **Unit Settings**. For details, see **Table 1-41**.	**HA scheduling**

Parameter	Description	Example
Scheduling Priority	Resource scheduling priority for service deployment. When **Service Resource Pool Settings** is set to **Dedicated resource pool**, set this parameter. The platform handles jobs by prioritizing them from highest to lowest. Range: 1 (lowest) to 3 (highest). If multiple jobs have the same priority, they are scheduled in the order they were submitted. The final schedule depends on available resources. When resources are enough and priorities match, the first jobs submitted are scheduled first.	**1**

Configuring Model Settings

The following table describes the model settings.

Table 1-14 Model settings

Category	Parameter	Description	Example
Model Source	Platform asset	When deploying a real-time service, you can select either **Platform asset** or **Custom Model** as the model source. When selecting **Platform asset**, the model originates from the ModelArts asset center. Click a card to select a model. You can choose from either **Preset Models** or **My Models**: • **Preset Models**: These are inference model assets built into ModelArts that you can select and use directly. • **My Models**: These are models generated by importing a local model or by completing model pre-training or model fine-tuning on ModelArts. Select your model and click **OK**. Supported model assets depend on the resource pool specifications. Select a dedicated resource pool or switch to the public resource pool and try again.	**Platform asset**
	Custom Model	When selecting a custom model, you can select the model storage type. Set the model storage address and mount path. Supported storage types for custom models: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. For details about the parameters, see **Storage Mounting**.	Custom Model

Configuring Prefill Inference Units

For multi-role PD co-location, set the inference service deployment mode to **Multi-role**.

Multi-role: Requires configuring multiple inference units as needed, with each unit corresponding to a specific role within the inference deployment instance. Adding or scaling roles requires adding new units. Multiple units are combined to form a complete inference deployment instance, suitable for scenarios such as PD co-location or disaggregation.

For details about the inference unit parameters, see the table below. During the configuration, you can click **Preview Deployment** in the upper right corner to view the topology of the current deployment, as shown in **Viewing the Service Deployment Topology**.

You can click **Clone** to copy the configuration of an existing inference unit, or click **Delete** to delete an inference unit that is no longer used.

Table 1-15 Inference unit parameters

Parameter	Description	Example
role-0	The default value is **role-0**, which can be modified. Enter up to 16 characters. Only lowercase letters, digits, and hyphens (-) are supported, and it must start and end with a letter or digit.	**role-prefill**
Unit Replicas	For multi-role separation, you can set or adjust the number of unit replicas to improve the unit throughput and response speed. The value ranges from 1 to 100. Set this parameter based on the number of concurrent requests.	**2**
Specification Type > Preset	Public resource pools only support preset specifications. Dedicated resource pools support preset specifications and custom specifications. Select an available unit instance specification and set the number of instances per replica. Total resource requirements of the inference unit = Unit instance specifications x Number of resource instances. For example, if the unit instance specification provides eight accelerators and you need 32 in total, enter **4**. **Prefill instance type: High-compute GPU/NPU specifications recommended** for compute-intensive prefill. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead.	/

Parameter	Description	Example
Specification Type > Custom	Only dedicated resource pools support custom specifications. Choose a custom specification if the preset ones do not fit your needs. GPU virtualization is enabled for this resource pool. Resources will be strictly allocated based on request volume. Configure specific vCPUs and memory. For details, see **Viewing Details About a Real-Time Service**. • CPU cores: The value must be at least 0.01 with exactly two decimal places and should not exceed the total cores available in the resource pool. • Memory: The memory value must be an integer of at least 4 and should not exceed the total memory available in the resource pool. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead. When deploying a real-time service, you may select a heterogeneous dedicated resource pool—one that contains multiple architectures or specifications. For example, a resource pool might include node pool 1 (x86 architecture, CPU-only, 8 vCPUs, 32 GB), node pool 2 (x86 architecture, CPU-only, 8 vCPUs, 64 GB), and node pool 3 (Arm architecture, NPU-accelerated, 192 vCPUs, 1536 GB). When deploying a real-time service using such a heterogeneous resource pool, you can first select the node pool specifications, for example, node pool 1 with CPU-only specifications, and then select the job specifications required for deploying the service, such as the number of CPUs, memory, or accelerators required for deploying the real-time service. If you have node pools with the same specification, like node pool 1 and node pool 2, and you choose node pool 1 during setup, the service may still be deployed in either node pool 1 or node pool 2. This depends on which pool has better resource usage at the time. When deploying multiple inference units, you cannot use heterogeneous node pools. All inference units must be in the same node pool.	/

Parameter	Description	Example
Image Type	Select the source of the inference image. • Preset image: Use images from ModelArts asset management. You can tag them with details like supported specifications and frameworks for easy management. For details, see **ModelArts Images**. • Custom image: Select your custom image. For details about how to create a custom image, see **Creating a Custom Image for a Model**. Use your own image via any of the following options: – **SoftWare Repository for Container (SWR)**: SWR is a secure, reliable, and easy-to-use container image management service that supports full lifecycle management of container images. – Registered image: Select an image registered with ModelArts. • Image URL: Enter the image path of a custom image. • **Pre-warmed image**: Select an image pre-warmed in dedicated resource pool in advance. If the image warmup task is done and the status is normal, deployment speeds up automatically. The selected image path and the service to be deployed must be in the same region.	/

Parameter	Description	Example
Environment Variables (Optional)	Inject environment variables into the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables. Environment variables are key-value pairs. For example, the key is **A** and the value is **AAAA**. Only letters, digits, underscores (_), hyphens (-), and periods (.) are supported. They cannot start with a digit and cannot exceed 64 characters. The value of an environment variable cannot be in HTML format, such as <p>, <^>, and <...>. You can upload an Excel file to batch import environment variables. Only .xlsx and .xls files are supported, and up to 100 parameters can be uploaded. To ensure correct parsing, fill in the file strictly according to the template format. To download the template, click **Download Template**. Click **Local Upload**. In the dialog box that is displayed, click **Add File** to import the local Excel file. Check the parsed environment variable key values. If the check result is **To be modified**, modify the values as required and upload the file again. After the check result is **Passed**, select the key values and click **OK** to complete the batch upload of environment variables.	/
Mount File Storage	Supports mounting file storage and specifying artifact dump path. Supported storage types for file storage: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. Supported storage type for artifact dumping: OBS parallel file system You can add up to 15 storage paths for file storage and 10 storage paths for artifact dump. Artifact dump cannot share the same OBS parallel file system with file or model mounting. For details about the storage type, storage address, and mount path, see **Storage Mounting**.	**Object Storage Service – Bucket**

Parameter	Description	Example
Health Check	Configure probes to monitor the model health. It can only be configured when the health check API is configured in the inference image. Otherwise, the model deployment fails. The following probes are supported: • **Startup Probe**: This probe checks if the instance has started. If a startup probe is configured, the liveness or readiness probe is not executed until the startup probe is successful, allowing sufficient time for the application to complete initialization. If the startup probe fails, the instance is restarted. If startup probe is not configured, the service status changes to success immediately after the service is scheduled. The service may be in the **Running** state and the prediction cannot be performed because the model is being loaded. • **Readiness Probe**: This probe verifies whether the instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. For details about the three types of probes involved in health check, see **Real-Time Service Health Check**.	/
Boot Command	Set the service boot command.	/
More Settings > Automatic Rebuild	When enabled, if a pod restarts due to deployment configuration changes or failures, the platform will automatically rebuild it using the selected policy. If disabled, the platform will not intervene. For details, see **Auto Rebuild upon a Real-Time Service Fault**.	/
More Settings > Auto Restart	When enabled, if an NPU, switch, or hardware fault is detected, services on the faulty node are automatically rescheduled. Some capabilities are only supported by Snt9b and Snt9b2 resources. For details, see **Auto Restart upon a Real-Time Service Fault**. To ensure service continuity, configure multiple instances.	/

Parameter	Description	Example
More Settings > Graceful Shutdown	Enabling this feature allows you to configure shutdown timeouts and commands. This prevents in-progress requests from being forcefully interrupted, thereby enhancing the availability and stability of the system. If you have configured health checks and set a large sleep value in this command, it will result in a longer restart or stop time for the container when the health check fails. • **Shutdown Timeout (s)**: This parameter indicates the maximum length of time that can pass between when a Pod receives a stop signal to when it is forcefully stopped. It is used for the Pod to perform cleanup operations (such as closing connections, releasing resources, and saving states). • **Shutdown Command**: The shutdown command is triggered when the container receives a stop signal, but it must be completed within the grace period or the shutdown times out. If this happens, the container will be forcefully stopped. You can use this command to perform operations such as closing database connections, releasing file handles, and stopping child processes.	/

Parameter	Description	Example
More Settings > Affinity Scheduling	You can configure the node affinity type and strength to flexibly schedule workloads in a resource pool. If no nodes are specified, the pods will be randomly scheduled according to the default cluster scheduling policy. After this function is enabled, you can refine the pod deployment policy. • Node affinity (strong): The pod must be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node affinity (weak): The system will try to place the pod on the specified node, but it is not guaranteed. • Node anti-affinity (strong): The pod must not be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node anti-affinity (weak): The system will try to avoid deploying the pod on the specified node, but it is not guaranteed In the **Add Node** list, select the nodes that meet the preceding configuration rules. When you choose a pre-warmed model, only the pre-warmed nodes appear on the affinity scheduling page. Unwarmed nodes do not show. A message appears stating: The selected model is pre-warmed and will deploy automatically on the best node. Specifying a node might cause warmup to fail.	/
More Settings > Container User ID	If this option is selected, enter the user ID and user group ID (optional).	/

Parameter	Description	Example
Authentication Credential	Displays when **System Log Reporting** is selected in **Table 1-5**. A secret is used to verify identity and authorize access. In the information security and identity authentication fields, a secret is a key mechanism to ensure that only authorized users can access the system, resources, or services. • If the version of CCE Container Storage (Everest) in your dedicated resource pool is v2.4.204 or later and the cluster version is v1.28 or later, temporary credentials are enabled by default (no AK/SK is required, ensuring higher security). • If your CCE Container Storage (Everest) version is too low or the cluster does not support temporary credentials, you must mount the credentials using DEW. When using DEW secrets to mount storage, you must include 'accessKeyId' and 'secretAccessKey' (corresponding to your AK and SK, respectively). Ensure the information provided is correct, or the function may not work as expected. To create a secret, click **Create Secret** to go to the DEW console. For details, see **Creating a Secret**. Set the AK/SK in the **Secret key/value** tab. Enter **accessKeyId** and **secretAccessKey** in the **Key** column. To obtain the values, go to the DEW console, and choose **My Credentials** > **Access Keys**.	/

Configuring Decode Inference Units

Click **Add Inference Unit** or click **Clone** to copy the existing inference unit settings and configure the decode inference units.

For details about the inference unit parameters, see the table below. During the configuration, you can click **Preview Deployment** in the upper right corner to view the topology of the current deployment, as shown in **Viewing the Service Deployment Topology**.

Table 1-16 Inference unit parameters

Parameter	Description	Example
role-1.	The default value is **role-1**, which can be modified. Enter up to 16 characters. Only lowercase letters, digits, and hyphens (-) are supported, and it must start and end with a letter or digit.	**role-decode**
Unit Replicas	For multi-role separation, you can set or adjust the number of unit replicas to improve the unit throughput and response speed. The value ranges from 1 to 100. Set this parameter based on the number of concurrent requests.	**2**
Specification Type > Preset	Public resource pools only support preset specifications. Dedicated resource pools support preset specifications and custom specifications. Select an available unit instance specification and set the number of instances per replica. Total resource requirements of the inference unit = Unit instance specifications x Number of resource instances. For example, if the unit instance specification provides eight accelerators and you need 32 in total, enter **4**. **For decode inference units, you are advised to select low-compute CPU/NPU specifications** to better adapt to decode streaming generation. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead.	/
Specification Type > Custom	Only dedicated resource pools support custom specifications. Choose a custom specification if the preset ones do not fit your needs. GPU virtualization is enabled for this resource pool. Resources will be strictly allocated based on request volume. Configure specific vCPUs and memory. For details, see **Viewing Details About a Real-Time Service**. • CPU cores: The value must be at least 0.01 with exactly two decimal places and should not exceed the total cores available in the resource pool. • Memory: The memory value must be an integer of at least 4 and should not exceed the total memory available in the resource pool. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead. When deploying multiple inference units, you cannot use heterogeneous node pools. All inference units must be in the same node pool.	/

Parameter	Description	Example
Image Type	Select the source of the inference image. • Preset image: Use images from ModelArts asset management. You can tag them with details like supported specifications and frameworks for easy management. For details, see **ModelArts Images**. • Custom image: Select your custom image. For details about how to create a custom image, see **Creating a Custom Image for a Model**. Use your own image via any of the following options: – **SoftWare Repository for Container (SWR)**: SWR is a secure, reliable, and easy-to-use container image management service that supports full lifecycle management of container images. – Registered image: Select an image registered with ModelArts. • Image URL: Enter the image path of a custom image. • **Pre-warmed image**: Select an image pre-warmed in dedicated resource pool in advance. If the image warmup task is done and the status is normal, deployment speeds up automatically. The selected image path and the service to be deployed must be in the same region.	**Custom Images > SWR**
Environment Variables (Optional)	Inject environment variables into the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables. Environment variables are key-value pairs. For example, the key is **A** and the value is **AAAA**. Only letters, digits, underscores (_), hyphens (-), and periods (.) are supported. They cannot start with a digit and cannot exceed 64 characters. The value cannot contain HTML tags, such as <p>, <^>, and <...>. You can upload an Excel file to batch import environment variables. Only .xlsx and .xls files are supported, and up to 100 parameters can be uploaded. To ensure correct parsing, fill in the file strictly according to the template format. To download the template, click **Download Template**. Click **Local Upload**. In the dialog box that is displayed, click **Add File** to import the local Excel file. Check the parsed environment variable key values. If the check result is **To be modified**, modify the values as required and upload the file again. After the check result is **Passed**, select the key values and click **OK** to complete the batch upload of environment variables.	/

Parameter	Description	Example
Mount File Storage	Supports mounting file storage and specifying artifact dump path. Supported storage types for file storage: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. Supported storage type for artifact dumping: OBS parallel file system You can add up to 15 storage paths for file storage and 10 storage paths for artifact dump. Artifact dump cannot share the same OBS parallel file system with file or model mounting. For details about the storage type, storage address, and mount path, see **Storage Mounting**.	**Object Storage Service – Bucket**
Health Check	Configure probes to monitor the model health. It can only be configured when the health check API is configured in the inference image. Otherwise, the model deployment fails. The following probes are supported: • **Startup Probe**: This probe checks if the application instance has started. If a startup probe is provided, all other probes are disabled until it succeeds. If the startup probe fails, the instance is restarted. If startup probe is not configured, the service status changes to success immediately after the service is scheduled. The service may be in the **Running** state and the prediction cannot be performed because the model is being loaded. • **Readiness Probe**: This probe verifies whether the application instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. For details about the three types of probes involved in health check, see **Real-Time Service Health Check**.	/
Boot Command	Set the service boot command.	/

Parameter	Description	Example
More Settings > Automatic Rebuild	When enabled, if a pod restarts due to deployment configuration changes or failures, the platform will automatically rebuild it using the selected policy. If disabled, the platform will not intervene. For details, see **Auto Rebuild upon a Real-Time Service Fault**.	/
More Settings > Auto Restart	When enabled, if an NPU, switch, or hardware fault is detected, services on the faulty node are automatically rescheduled. Some capabilities are only supported by Snt9b and Snt9b2 resources. For details, see **Auto Restart upon a Real-Time Service Fault**. To ensure service continuity, configure multiple instances.	/
More Settings > Graceful Shutdown	Enabling this feature allows you to configure shutdown timeouts and commands. This prevents in-progress requests from being forcefully interrupted, thereby enhancing the availability and stability of the system. If you have configured health checks and set a large sleep value in this command, it will result in a longer restart or stop time for the container when the health check fails. • **Shutdown Timeout (s)**: This parameter indicates the maximum length of time that can pass between when a Pod receives a stop signal to when it is forcefully stopped. It is used for the Pod to perform cleanup operations (such as closing connections, releasing resources, and saving states). • **Shutdown Command**: The shutdown command is triggered when the container receives a stop signal, but it must be completed within the grace period or the shutdown times out. If this happens, the container will be forcefully stopped. You can use this command to perform operations such as closing database connections, releasing file handles, and stopping child processes.	/

Parameter	Description	Example
More Settings > Affinity Scheduling	You can configure the node affinity type and strength to flexibly schedule workloads in a resource pool. If no nodes are specified, the pods will be randomly scheduled according to the default cluster scheduling policy. After this function is enabled, you can refine the pod deployment policy. • Node affinity (strong): The pod must be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node affinity (weak): The system will try to place the pod on the specified node, but it is not guaranteed. • Node anti-affinity (strong): The pod must not be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node anti-affinity (weak): The system will try to avoid deploying the pod on the specified node, but it is not guaranteed In the **Add Node** list, select the nodes that meet the preceding configuration rules. When you choose a pre-warmed model, only the pre-warmed nodes appear on the affinity scheduling page. Unwarmed nodes do not show. A message appears stating: The selected model is pre-warmed and will deploy automatically on the best node. Specifying a node might cause warmup to fail.	/
More Settings > Container User ID	If this option is selected, enter the user ID and user group ID (optional).	/

Parameter	Description	Example
Authentication Credential	Displays when OBS is used for mounting and local storage acceleration is not enabled, or **System Log Reporting** is selected in **Table 1-5**. A secret is used to verify identity and authorize access. In the information security and identity authentication fields, a secret is a key mechanism to ensure that only authorized users can access the system, resources, or services. • If the version of CCE Container Storage (Everest) in your dedicated resource pool is v2.4.204 or later and the cluster version is v1.28 or later, temporary credentials are enabled by default (no AK/SK is required, ensuring higher security). • If your CCE Container Storage (Everest) version is too low or the cluster does not support temporary credentials, you must mount the credentials using DEW. When using DEW secrets to mount storage, you must include 'accessKeyId' and 'secretAccessKey' (corresponding to your AK and SK, respectively). Ensure the information provided is correct, or the function may not work as expected. To create a secret, click **Create Secret** to go to the DEW console. For details, see **Creating a Secret**. Set the AK/SK in the **Secret key/value** tab. Enter **accessKeyId** and **secretAccessKey** in the **Key** column. To obtain the values, go to the DEW console, and choose **My Credentials** > **Access Keys**.	/

Configuring Deployment Management Settings

The following table describes the deployment management settings.

Table 1-17 Deployment management settings

Parameter	Description
Container Protocol	The network transmission protocol used by the container. Configure it according to the API definitions in your actual setup. If **Service Protocol** is set to **HTTP** or **HTTPS** in **Table 1-2**, **Container Protocol** can be set to **HTTP** or **HTTPS**. If **Service Protocol** is set to **WSS** or **WS** in **Table 1-2**, the container protocol is the same as the service protocol by default and is not displayed on the console.

Parameter	Description
Container Port	The port number that the image listens on. Requests are sent to the instance through this port. The port number must be the same as the port number in your image.
More Settings > Deployment Timeout (Minutes)	The wait time before a single service times out. This value includes both the deployment and startup time. Set this parameter properly. The system will cancel the deployment if it exceeds the time limit.
More Settings > Max Surge Replicas (%)	The maximum number of replicas that can be created above the target count during a rolling upgrade When percentages are used, the instance count is rounded up. Example: When **Max Surge Replicas (%)** is set to **1%** during a rolling upgrade of four instances, you can add one new instance at a time. When **Max Surge Replicas (%)** is set to **100%** during a rolling upgrade of two instances, you can add two new instances immediately. For details about rolling upgrade, see **Rolling Upgrades for Real-Time Service Deployment**.

Parameter	Description
More Settings > Max Unavailable Replicas (%)	The maximum number of replicas that can be unavailable relative to the target count during a rolling update When percentages are used, the instance count is rounded down. Example: • When **Max Unavailable Replicas (%)** is set to **1** during a rolling upgrade of four instances, four instances must remain available. An old instance can only be deleted after its replacement starts running. • When **Max Unavailable Replicas (%)** is set to **50** during a rolling upgrade of two instances, one old instance can be deleted immediately to release resources, but at least one instance must remain available. • If you deploy a service with **Max Unavailable Replicas (%)** at **20**, requiring five instances, and only one fails due to lack of resources while four start successfully, the deployment succeeds, and the service status becomes **Running**. If **Max Unavailable Replicas** equals the target replica count, there is a risk of downtime (**Min Available Replicas** = **Total Replicas** – **Max Unavailable Replicas**). Example: Setting **Max. Invalid Instances** to **100** allows the service stay running when an instance fails because of quick recovery. Yet, the service will not be able to perform predictions, potentially leading to interruptions. To ensure a hitless upgrade, set Max. Invalid Instances to 1%. (This requires extra resources. Make sure you have enough, or the update will fail.) For details about rolling upgrade, see **Rolling Upgrades for Real-Time Service Deployment**.

Configuring Advanced Settings

The following table describes the advanced settings.

Table 1-18 Advanced settings

Parameter	Description
Key Settings	Inference services support secret mounting. This uses encryption to protect sensitive data and securely mount it to containers. To protect your data, do not enter sensitive information in plaintext. • **Custom key**: Enter the key, key value, and mount path. • **DEW**: Data Encryption Workshop (DEW) is a comprehensive cloud data encryption service, including Key Management Service (KMS), Key Pair Service (KPS), and Dedicated Hardware Security Module (Dedicated HSM). To use DEW to configure the key, go to the DEW console to **create a key**, select the DEW key on the ModelArts console, and enter the mount path. Choose whether to select **Associate Image User Group ID**. The user group ID of the user who starts the image can be associated. After the association, the secret mounting security is improved.
Intelligent Routing Policy	If intelligent routing is enabled, set intelligent routing policies. The following policies are supported: • Round robin: Tasks are distributed sequentially to different nodes in order. This ensures an even distribution across the cluster and achieves load balancing. • Source IP hash: The system calculates hash values using client IP addresses to route requests from the same IP address to the same node. • Least connections: Requests are routed to the node with the fewest real-time connections. This method ensures that the load is distributed more evenly, improving service stability and response time. • Minimum time to first token: Requests are routed to the node with the lowest average time to first token. This method aims to minimize the wait time between receiving a request and starting its processing. This latency applies even when system resources are available immediately. • Overall loads: Requests are routed to the node with the lowest overall pressure, considering factors such as the number of connections, time to first token, and custom metrics. It directs new requests to less busy nodes to avoid resource waste and overloading. • SLO priority: Services are prioritized according to their assigned Service Level Objective (SLO) (0–3, where 0 is the highest). This ensures lower latency for higher-priority workloads. For details, see **Intelligent Routing Policy**.

Parameter	Description
Custom Metric Collection	If this function is enabled, you need to input the metric collection port details. Custom metrics will send data to AOM's Prometheus instances for viewing on the AOM console. To query custom metrics, see **Viewing Performance Metrics of a Real-Time Service on AOM** and **Viewing Performance Metrics of a Real-Time Service on ModelArts**. **Constraints** • ModelArts calls the HTTP API provided in the custom metric configuration every 10 seconds to obtain metric data. • The metric data text returned by the HTTP API provided in the custom metric configuration cannot exceed 32 KB. **Data Format of Custom Metrics** The format of custom metrics data must comply with the open metrics specifications. That is, the format of each metric must be: <*metric_name*>{<*tag_name*>=<*tag_value*>,...} <*sample_value*> [*timestamp_in_millisecond*] The following shows an example (the comment starts with #, which is optional): # HELP http_requests_total The total number of HTTP requests. # TYPE http_requests_total gauge html_http_requests_total{method="post",code="200"} 1656 1686660980680 html_http_requests_total{method="post",code="400"} 2 1686660980681
System Log Reporting	Displays only for NPU dedicated resource pools. When enabled, system logs are mounted to a fixed parallel file system for O&M engineers to analyze. System logs are stored for 30 days and will be automatically deleted after that. The mount path of system logs cannot be changed.

Confirming Settings

On the **Deploy Real-Time Service** > **Confirmation** page, confirm the configuration information and click **Confirm Deployment**.

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.

You can go to the real-time service list to check if the deployment is complete. Once the service status changes from **Deploying** to **Running**, the service is deployed.

NOTE

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

Follow-Up Operations

- For details about how to test a real-time service, compare the model effect in real time, and debug a real-time service using CloudShell, see **Testing a Real-Time Service**.
- For details about how to view real-time service details, see **Viewing Details About a Real-Time Service**.
- For details about how to modify, stop, and delete a real-time service, see **Managing the Lifecycle of a Real-Time Service Deployment**.

1.2.4 Deploying a Real-Time Inference Service Using Multi-Node PD Disaggregation

NOTE

Real-time services have two versions. The new version is recommended.

Overview

In ultra-large-scale LLM inference scenarios (10B to 100B+ parameters), traditional single-node deployment faces pain points such as insufficient VRAM, high time-to-first-token (TTFT) latency, low concurrent throughput, and poor resource utilization. Consequently, it cannot meet the high-concurrency, low-latency, and high-stability demands of enterprise-grade generative AI (text generation, multimodal generation).

Multi-node PD disaggregation is designed specifically for generative scenarios such as LLMs, text-to-image generation, and long-context dialogues. It is perfectly suited for the following core scenarios:

- Ultra-large-scale model inference (34B/70B/175B+ parameters): Scenarios where the VRAM of a single node cannot accommodate the complete model weights.
- High-concurrency request scenarios: Scenarios requiring boosted inference throughput and reduced TTFT.
- Resource utilization optimization: Scenarios that need to align with the differentiated computing demands of the prefill phase (compute-intensive) and the decode phase (concurrency-intensive).

Core Principles

Generative inference for LLMs is divided into two core phases, each with significantly different compute and VRAM requirements:

- Prefill: Processes the user's input prompt and calculates the input sequence's KV cache all at once. This phase requires high VRAM and high compute, but has short execution time and low concurrency.
- Decode: Generates the output token-by-token based on the KV cache. This phase requires low VRAM and low compute, but has long execution time and high concurrency.

Deployment mode comparison:

- Multi-node prefill-decode (PD) co-location (multi-role separation – co-location): PD co-location refers to deploying both the prefill and decode stages of LLM inference onto the same set of compute nodes (such as NPUs/GPUs) to share KV cache resources. This mode is suitable for resource-constrained scenarios or when architecture simplification is required. When using the vLLM framework in a co-location scenario, you are advised to set the first unit as the vLLM master node and place the remaining worker nodes in other units. For details about multi-node PD co-location, see **Deploying a Real-Time Inference Service Using Multi-Node PD Co-location**.
- **Multi-node PD disaggregation (multi-role separation – isolation)**: In the multi-role separation mode, the prefill and decode units are deployed on different physical nodes to achieve complete resource isolation. This mode delivers optimal performance but incurs higher costs. **This topic focuses on multi-node PD disaggregation.**

Based on the ModelArts multi-role separation deployment mode, the two phases are split across different nodes to achieve multi-node collaboration:

- Unit splitting: Creates two types of independent inference units, prefill units (high-compute nodes) and decode units (high-concurrency nodes).
- Intelligent routing: The platform automatically distributes requests. A user request is first processed by the prefill unit to generate the KV cache, and is then forwarded to the decode unit to complete the token-by-token generation.
- Resource isolation: The two types of units are scheduled and scaled independently, allowing resources to be matched precisely on demand.
- Multi-node collaboration: Relying on the distributed capabilities of dedicated resource pools, the multi-node cluster communicates via a high-speed internal network to complete cross-node PD task collaboration.

Constraints

- Deployment mode constraints: The multi-role separation mode must be selected, consisting of at least one prefill unit and one decode unit.
- Resource specification constraints: Supports dedicated resource pools (including heterogeneous ones) and public resource pools. Physical pools that have already created logical subpools are currently not supported. When deploying on a heterogeneous resource pool, the instance specifications of the prefill and decode units must match the specifications of the node pool.
- Image/Model requirements: The model image must be adapted to the multi-role separation architecture and support independent deployment of prefill and decode; otherwise, the deployment will fail. The model must also support KV cache separation.
- Call mode constraints: Both synchronous and asynchronous calls are supported.

Prerequisites

- You have prepared data as instructed in **Preparations**.
- Your account is not in arrears to ensure available resources for running services.

- You have configured the inference service information as instructed in **Configuring Service Information**.
- The current account has the real-time inference deployment permission and dedicated resource pool scheduling permission. For details, see **Configuring Agency Authorization for ModelArts with One Click**.

Notes

Resource pools allocate quota for real-time services even when they are **Abnormal** or **Stopped**. If the quota is insufficient and no more services can be deployed, delete some abnormal services to release resources.

- Quota calculation:

  The quota stays the same when you deploy real-time services with a dedicated resource pool. It only changes if you create, modify, or delete a resource pool.
- Usage metering:

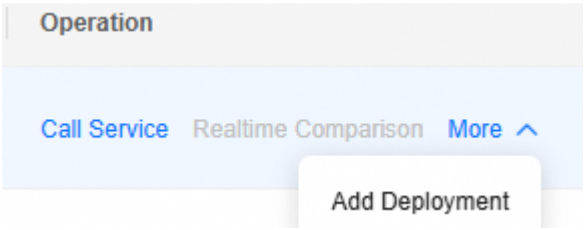
  Deploying real-time services in a dedicated resource pool is not metered. Only the usage of the dedicated resource pool itself is metered.
- Mounting SFS Turbo:

  Before mounting an SFS Turbo file system to a real-time service, **associate the dedicated resource pool network with the file system**.

Configuring Basic Information

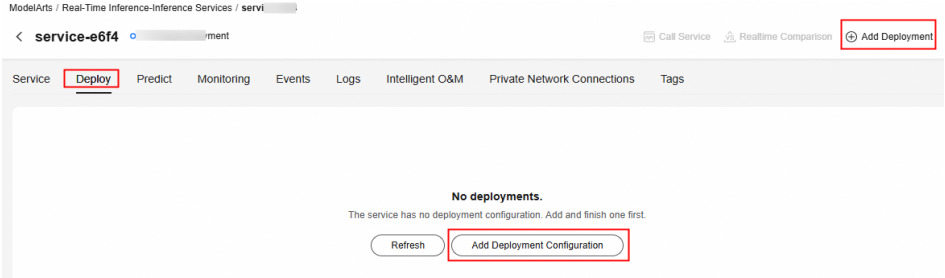
Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. On the service list page, choose **More** > **Add Deployment** on the right of the service name.

Figure 1-12 Adding a deployment



You can also click the service name to go to the service details page and configure deployment information on the **Deploy** tab page or by clicking **Add Deployment** in the upper right corner of the service details page.

Figure 1-13 Adding a deployment



On the **Deploy Real-Time Service** page, configure basic information, resource settings, model settings, unit settings, deployment management settings, and advanced settings.

The following table describes the basic information.

Table 1-19 Basic information

Parameter	Description	Example
Deployment Name	Name of the current deployment, which is used to identify and manage the deployment configuration of the real-time service. Enter a name as prompted. Only letters, digits, hyphens (-), and underscores (_) are supported. Max length: 128 characters	**service**
Description (Optional)	Brief description of the deployment.	/

Configuring Resource Settings

The following table describes the resource settings.

Table 1-20 Resource settings

Parameter	Description	Example
Resource Pool Type	Both dedicated resource pools and public resource pools support multi-role separation deployment. • **Public Resource Pool** Public resource pool for deploying the real-time service. The public resource pool supplies shared compute clusters assigned according to job parameters. Each job operates with its own isolated resources. This option offers cost-effective and flexible solutions for tasks like development and testing. Choosing the public resource pool might leave fewer resources available because of its limits. If this happens, join the queue and wait your turn. • **Dedicated Resource Pool** Dedicated resource pool for deploying the real-time service. The resources provided in a dedicated resource pool are exclusive and more controllable. Use dedicated resource pools for core production services to secure exclusive resources.	**Dedicated Resource Pool**

Parameter	Description	Example
Resource Pool	If **Resource Pool Type** is set to **Dedicated Resource Pool**, click **Select Resource Pool**, select the corresponding specifications in the dedicated resource pool specifications area, and click **OK**. The physical pools with logical subpools created are not supported. If no dedicated resource pool is available, create one. You can choose a heterogeneous resource pool for deploying real-time services. If you use a heterogeneous dedicated resource pool, ensure the real-time service's instance specifications match the pool's specifications.	/
Deployment Replicas	A deployment instance consists of units capable of independently completing an inference task. Specify the number of instance replicas for the deployment. The value ranges from **1** to **128**. If there is one replica, only one service instance runs. This is a standard single-node setup. With three replicas, three identical service instances run simultaneously. This spreads out requests, improves handling multiple tasks, and provides basic high availability.	2 to 4
Scheduling Policy	Two scheduling policies are available: **HA scheduling** and **Compact scheduling**. **HA scheduling** is enabled by default. • **HA scheduling**: Pods from different replicas will be distributed across different nodes as evenly as possible, while multiple pods under the same replica will prioritize scheduling to the same node to ensure high availability for inference services. If both **HA scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence. • **Compact scheduling**: Enable bin packing for cluster workloads. The scheduler will prioritize placing pods on nodes with higher resource consumption to reduce idle resource fragmentation and improve cluster resource utilization. When both **Compact scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence. Affinity scheduling is set in **More Settings** of the **Unit Settings**. For details, see **Table 1-41**.	**HA scheduling**

Parameter	Description	Example
Scheduling Priority	Resource scheduling priority for service deployment. When **Service Resource Pool Settings** is set to **Dedicated resource pool**, set this parameter. The platform handles jobs by prioritizing them from highest to lowest. Range: 1 (lowest) to 3 (highest). If multiple jobs have the same priority, they are scheduled in the order they were submitted. The final schedule depends on available resources. When resources are enough and priorities match, the first jobs submitted are scheduled first.	**1**

Configuring Model Settings

The following table describes the model settings.

Table 1-21 Model settings

Category	Parameter	Description	Example
Model Source	Platform asset	When deploying a real-time service, you can select either **Platform asset** or **Custom Model** as the model source. When selecting **Platform asset**, the model originates from the ModelArts asset center. Click a card to select a model. You can choose from either **Preset Models** or **My Models**: • **Preset Models**: These are inference model assets built into ModelArts that you can select and use directly. • **My Models**: These are models generated by importing a local model or by completing model pre-training or model fine-tuning on ModelArts. Select your model and click **OK**. Supported model assets depend on the resource pool specifications. Select a dedicated resource pool or switch to the public resource pool and try again.	**Platform asset**
	Custom Model	When selecting a custom model, you can select the model storage type. Set the model storage address and mount path. Supported storage types for custom models: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. For details about the parameters, see **Storage Mounting**.	**Custom Model**

Configuring Prefill Inference Units

For multi-role PD disaggregation, set the inference service deployment mode to **Multi-role**.

Multi-role: Requires configuring multiple inference units as needed, with each unit corresponding to a specific role within the inference deployment instance. Adding or scaling roles requires adding new units. Multiple units are combined to form a complete inference deployment instance, suitable for scenarios such as PD co-location or disaggregation.

xPyD general ratio rules

x = Number of prefill units; y = Number of decode units; x + y = Total number of units

Example: 1P2D (1 prefill unit + 2 decode units) is the standard configuration for mainstream online production environments.

For details about the inference unit parameters, see the table below. During the configuration, you can click **Preview Deployment** in the upper right corner to view the topology of the current deployment, as shown in **Viewing the Service Deployment Topology**.

You can click **Clone** to copy the configuration of an existing inference unit, or click **Delete** to delete an inference unit that is no longer used.

Table 1-22 Prefill inference unit parameters

Parameter	Description	Example
role-0	The default value is **role-0**, which can be modified. Enter up to 16 characters. Only lowercase letters, digits, and hyphens (-) are supported, and it must start and end with a letter or digit.	**role-prefill**
Unit Replicas	For multi-role separation, you can set or adjust the number of unit replicas to improve the unit throughput and response speed. The value ranges from 1 to 100. Set this parameter based on the number of concurrent requests.	**1**

Parameter	Description	Example
Specification Type > Preset	Public resource pools only support preset specifications. Dedicated resource pools support preset specifications and custom specifications. Select an available unit instance specification and set the number of instances per replica. Total resource requirements of the inference unit = Unit instance specifications x Number of resource instances. For example, if the unit instance specification provides eight accelerators and you need 32 in total, enter **4**. **Prefill instance type: High-compute GPU/NPU specifications recommended** for compute-intensive prefill. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead.	/

Parameter	Description	Example
Specification Type > Custom	Only dedicated resource pools support custom specifications. Choose a custom specification if the preset ones do not fit your needs. GPU virtualization is enabled for this resource pool. Resources will be strictly allocated based on request volume. Configure specific vCPUs and memory. For details, see **Viewing Details About a Real-Time Service**. • CPU cores: The value must be at least 0.01 with exactly two decimal places and should not exceed the total cores available in the resource pool. • Memory: The memory value must be an integer of at least 4 and should not exceed the total memory available in the resource pool. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead. When deploying a real-time service, you may select a heterogeneous dedicated resource pool—one that contains multiple architectures or specifications. For example, a resource pool might include node pool 1 (x86 architecture, CPU-only, 8 vCPUs, 32 GB), node pool 2 (x86 architecture, CPU-only, 8 vCPUs, 64 GB), and node pool 3 (Arm architecture, NPU-accelerated, 192 vCPUs, 1536 GB). When deploying a real-time service using such a heterogeneous resource pool, you can first select the node pool specifications, for example, node pool 1 with CPU-only specifications, and then select the job specifications required for deploying the service, such as the number of CPUs, memory, or accelerators required for deploying the real-time service. If you have node pools with the same specification, like node pool 1 and node pool 2, and you choose node pool 1 during setup, the service may still be deployed in either node pool 1 or node pool 2. This depends on which pool has better resource usage at the time. When deploying multiple inference units, you cannot use heterogeneous node pools. All inference units must be in the same node pool.	/

Parameter	Description	Example
Image Type	Select the source of the inference image. • Preset image: Use images from ModelArts asset management. You can tag them with details like supported specifications and frameworks for easy management. For details, see **ModelArts Images**. • Custom image: Select your custom image. For details about how to create a custom image, see **Creating a Custom Image for a Model**. Use your own image via any of the following options: – **SoftWare Repository for Container (SWR)**: SWR is a secure, reliable, and easy-to-use container image management service that supports full lifecycle management of container images. – Registered image: Select an image registered with ModelArts. • Image URL: Enter the image path of a custom image. • **Pre-warmed image**: Select an image pre-warmed in dedicated resource pool in advance. If the image warmup task is done and the status is normal, deployment speeds up automatically. The selected image path and the service to be deployed must be in the same region.	/

Parameter	Description	Example
Environment Variables (Optional)	Inject environment variables into the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables. Environment variables are key-value pairs. For example, the key is **A** and the value is **AAAA**. Only letters, digits, underscores (_), hyphens (-), and periods (.) are supported. They cannot start with a digit and cannot exceed 64 characters. The value of an environment variable cannot be in HTML format, such as <p>, <^>, and <...>. You can upload an Excel file to batch import environment variables. Only .xlsx and .xls files are supported, and up to 100 parameters can be uploaded. To ensure correct parsing, fill in the file strictly according to the template format. To download the template, click **Download Template**. Click **Local Upload**. In the dialog box that is displayed, click **Add File** to import the local Excel file. Check the parsed environment variable key values. If the check result is **To be modified**, modify the values as required and upload the file again. After the check result is **Passed**, select the key values and click **OK** to complete the batch upload of environment variables.	/
Mount File Storage	Supports mounting file storage and specifying artifact dump path. Supported storage types for file storage: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. Supported storage type for artifact dumping: OBS parallel file system You can add up to 15 storage paths for file storage and 10 storage paths for artifact dump. Artifact dump cannot share the same OBS parallel file system with file or model mounting. For details about the storage type, storage address, and mount path, see **Storage Mounting**.	/

Parameter	Description	Example
Health Check	Configure probes to monitor the model health. It can only be configured when the health check API is configured in the inference image. Otherwise, the model deployment fails. The following probes are supported: • **Startup Probe**: This probe checks if the instance has started. If a startup probe is configured, the liveness or readiness probe is not executed until the startup probe is successful, allowing sufficient time for the application to complete initialization. If the startup probe fails, the instance is restarted. If startup probe is not configured, the service status changes to success immediately after the service is scheduled. The service may be in the **Running** state and the prediction cannot be performed because the model is being loaded. • **Readiness Probe**: This probe verifies whether the instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. For details about the three types of probes involved in health check, see **Real-Time Service Health Check**.	/
Boot Command	Set the service boot command.	/
More Settings > Automatic Rebuild	When enabled, if a pod restarts due to deployment configuration changes or failures, the platform will automatically rebuild it using the selected policy. If disabled, the platform will not intervene. For details, see **Auto Rebuild upon a Real-Time Service Fault**.	/
More Settings > Auto Restart	When enabled, if an NPU, switch, or hardware fault is detected, services on the faulty node are automatically rescheduled. Some capabilities are only supported by Snt9b and Snt9b2 resources. For details, see **Auto Restart upon a Real-Time Service Fault**. To ensure service continuity, configure multiple instances.	/

Parameter	Description	Example
More Settings > Graceful Shutdown	Enabling this feature allows you to configure shutdown timeouts and commands. This prevents in-progress requests from being forcefully interrupted, thereby enhancing the availability and stability of the system. If you have configured health checks and set a large sleep value in this command, it will result in a longer restart or stop time for the container when the health check fails. • **Shutdown Timeout (s)**: This parameter indicates the maximum length of time that can pass between when a Pod receives a stop signal to when it is forcefully stopped. It is used for the Pod to perform cleanup operations (such as closing connections, releasing resources, and saving states). • **Shutdown Command**: The shutdown command is triggered when the container receives a stop signal, but it must be completed within the grace period or the shutdown times out. If this happens, the container will be forcefully stopped. You can use this command to perform operations such as closing database connections, releasing file handles, and stopping child processes.	/

Parameter	Description	Example
More Settings > Affinity Scheduling	You can configure the node affinity type and strength to flexibly schedule workloads in a resource pool. If no nodes are specified, the pods will be randomly scheduled according to the default cluster scheduling policy. After this function is enabled, you can refine the pod deployment policy. • Node affinity (strong): The pod must be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node affinity (weak): The system will try to place the pod on the specified node, but it is not guaranteed. • Node anti-affinity (strong): The pod must not be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node anti-affinity (weak): The system will try to avoid deploying the pod on the specified node, but it is not guaranteed In the **Add Node** list, select the nodes that meet the preceding configuration rules. When you choose a pre-warmed model, only the pre-warmed nodes appear on the affinity scheduling page. Unwarmed nodes do not show. A message appears stating: The selected model is pre-warmed and will deploy automatically on the best node. Specifying a node might cause warmup to fail.	/
More Settings > Container User ID	If this option is selected, enter the user ID and user group ID (optional).	/

Parameter	Description	Example
Authentication Credential	Displays when OBS is used for mounting and local storage acceleration is not enabled, or **System Log Reporting** is selected in **Table 1-5**. A secret is used to verify identity and authorize access. In the information security and identity authentication fields, a secret is a key mechanism to ensure that only authorized users can access the system, resources, or services. • If the version of CCE Container Storage (Everest) in your dedicated resource pool is v2.4.204 or later and the cluster version is v1.28 or later, temporary credentials are enabled by default (no AK/SK is required, ensuring higher security). • If your CCE Container Storage (Everest) version is too low or the cluster does not support temporary credentials, you must mount the credentials using DEW. When using DEW secrets to mount storage, you must include 'accessKeyId' and 'secretAccessKey' (corresponding to your AK and SK, respectively). Ensure the information provided is correct, or the function may not work as expected. To create a secret, click **Create Secret** to go to the DEW console. For details, see **Creating a Secret**. Set the AK/SK in the **Secret key/value** tab. Enter **accessKeyId** and **secretAccessKey** in the **Key** column. To obtain the values, go to the DEW console, and choose **My Credentials** > **Access Keys**.	/

Configuring Decode Inference Units

Click **Add Inference Unit** or click **Clone** to copy the existing inference unit settings and configure the decode inference units.

For details about the inference unit parameters, see the table below. During the configuration, you can click **Preview Deployment** in the upper right corner to view the topology of the current deployment, as shown in **Viewing the Service Deployment Topology**.

Table 1-23 Decode inference unit parameters

Parameter	Description	Example
role-1.	The default value is **role-1**, which can be modified. Enter up to 16 characters. Only lowercase letters, digits, and hyphens (-) are supported, and it must start and end with a letter or digit.	**role-decode**
Unit Replicas	For multi-role separation, you can set or adjust the number of unit replicas to improve the unit throughput and response speed. The value ranges from 1 to 100. Set this parameter based on the number of concurrent requests.	**2**
Specification Type > Preset	Dedicated resource pools support preset specifications and custom specifications. Select an available unit instance specification and set the number of instances per replica. Total resource requirements of the inference unit = Unit instance specifications x Number of resource instances. For example, if the unit instance specification provides eight accelerators and you need 32 in total, enter **4**. **For decode inference units, you are advised to select low-compute CPU/NPU specifications** to better adapt to decode streaming generation. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead.	/
Specification Type > Custom	Only dedicated resource pools support custom specifications. Choose a custom specification if the preset ones do not fit your needs. GPU virtualization is enabled for this resource pool. Resources will be strictly allocated based on request volume. Configure specific vCPUs and memory. For details, see **Viewing Details About a Real-Time Service**. • CPU cores: The value must be at least 0.01 with exactly two decimal places and should not exceed the total cores available in the resource pool. • Memory: The memory value must be an integer of at least 4 and should not exceed the total memory available in the resource pool. **NOTE** Actual consumption will be slightly higher than the selected specification due to system overhead. When deploying multiple inference units, you cannot use heterogeneous node pools. All inference units must be in the same node pool.	/

Parameter	Description	Example
Image Type	Select the source of the inference image. • Preset image: Use images from ModelArts asset management. You can tag them with details like supported specifications and frameworks for easy management. For details, see **ModelArts Images**. • Custom image: Select your custom image. For details about how to create a custom image, see **Creating a Custom Image for a Model**. Use your own image via any of the following options: – **SoftWare Repository for Container (SWR)**: SWR is a secure, reliable, and easy-to-use container image management service that supports full lifecycle management of container images. – Registered image: Select an image registered with ModelArts. • Image URL: Enter the image path of a custom image. • **Pre-warmed image**: Select an image pre-warmed in dedicated resource pool in advance. If the image warmup task is done and the status is normal, deployment speeds up automatically. The selected image path and the service to be deployed must be in the same region.	**Custom Images > SWR**

Parameter	Description	Example
Environment Variables (Optional)	Inject environment variables into the pod. To ensure data security, do not enter sensitive information, such as plaintext passwords, in environment variables. Environment variables are key-value pairs. For example, the key is **A** and the value is **AAAA**. Only letters, digits, underscores (_), hyphens (-), and periods (.) are supported. They cannot start with a digit and cannot exceed 64 characters. The value of an environment variable cannot be in HTML format, such as <p>, <^>, and <...>. You can upload an Excel file to batch import environment variables. Only .xlsx and .xls files are supported, and up to 100 parameters can be uploaded. To ensure correct parsing, fill in the file strictly according to the template format. To download the template, click **Download Template**. Click **Local Upload**. In the dialog box that is displayed, click **Add File** to import the local Excel file. Check the parsed environment variable key values. If the check result is **To be modified**, modify the values as required and upload the file again. After the check result is **Passed**, select the key values and click **OK** to complete the batch upload of environment variables.	/
Mount File Storage	Supports mounting file storage and specifying artifact dump path. Supported storage types for file storage: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models. Supported storage type for artifact dumping: OBS parallel file system You can add up to 15 storage paths for file storage and 10 storage paths for artifact dump. Artifact dump cannot share the same OBS parallel file system with file or model mounting. For details about the storage type, storage address, and mount path, see **Storage Mounting**.	**Object Storage Service – Bucket**

Parameter	Description	Example
Health Check	Configure probes to monitor the model health. It can only be configured when the health check API is configured in the inference image. Otherwise, the model deployment fails. The following probes are supported: • **Startup Probe**: This probe checks if the application instance has started. If a startup probe is configured, the liveness or readiness probe is not executed until the startup probe is successful, allowing sufficient time for the application to complete initialization. If the startup probe fails, the instance is restarted. If startup probe is not configured, the service status changes to success immediately after the service is scheduled. The service may be in the **Running** state and the prediction cannot be performed because the model is being loaded. • **Readiness Probe**: This probe verifies whether the application instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. For details about the three types of probes involved in health check, see **Real-Time Service Health Check**.	/
Boot Command	Set the service boot command.	/
More Settings > Automatic Rebuild	When enabled, if a pod restarts due to deployment configuration changes or failures, the platform will automatically rebuild it using the selected policy. If disabled, the platform will not intervene. For details, see **Auto Rebuild upon a Real-Time Service Fault**.	/
More Settings > Auto Restart	When enabled, if an NPU, switch, or hardware fault is detected, services on the faulty node are automatically rescheduled. Some capabilities are only supported by Snt9b and Snt9b2 resources. For details, see **Auto Restart upon a Real-Time Service Fault**. To ensure service continuity, configure multiple instances.	/

Parameter	Description	Example
More Settings > Graceful Shutdown	Enabling this feature allows you to configure shutdown timeouts and commands. This prevents in-progress requests from being forcefully interrupted, thereby enhancing the availability and stability of the system. If you have configured health checks and set a large sleep value in this command, it will result in a longer restart or stop time for the container when the health check fails. • **Shutdown Timeout (s)**: This parameter indicates the maximum length of time that can pass between when a Pod receives a stop signal to when it is forcefully stopped. It is used for the Pod to perform cleanup operations (such as closing connections, releasing resources, and saving states). • **Shutdown Command**: The shutdown command is triggered when the container receives a stop signal, but it must be completed within the grace period or the shutdown times out. If this happens, the container will be forcefully stopped. You can use this command to perform operations such as closing database connections, releasing file handles, and stopping child processes.	/

Parameter	Description	Example
More Settings > Affinity Scheduling	You can configure the node affinity type and strength to flexibly schedule workloads in a resource pool. If no nodes are specified, the pods will be randomly scheduled according to the default cluster scheduling policy. After this function is enabled, you can refine the pod deployment policy. • Node affinity (strong): The pod must be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node affinity (weak): The system will try to place the pod on the specified node, but it is not guaranteed. • Node anti-affinity (strong): The pod must not be scheduled onto the specified node; otherwise, scheduling will not proceed. • Node anti-affinity (weak): The system will try to avoid deploying the pod on the specified node, but it is not guaranteed In the **Add Node** list, select the nodes that meet the preceding configuration rules. When you choose a pre-warmed model, only the pre-warmed nodes appear on the affinity scheduling page. Unwarmed nodes do not show. A message appears stating: The selected model is pre-warmed and will deploy automatically on the best node. Specifying a node might cause warmup to fail.	/
More Settings > Container User ID	If this option is selected, enter the user ID and user group ID (optional).	/

Parameter	Description	Example
Authentication Credential	Displays when OBS is used for mounting and local storage acceleration is not enabled, or **System Log Reporting** is selected in **Table 1-5**. A secret is used to verify identity and authorize access. In the information security and identity authentication fields, a secret is a key mechanism to ensure that only authorized users can access the system, resources, or services. • If the version of CCE Container Storage (Everest) in your dedicated resource pool is v2.4.204 or later and the cluster version is v1.28 or later, temporary credentials are enabled by default (no AK/SK is required, ensuring higher security). • If your CCE Container Storage (Everest) version is too low or the cluster does not support temporary credentials, you must mount the credentials using DEW. When using DEW secrets to mount storage, you must include 'accessKeyId' and 'secretAccessKey' (corresponding to your AK and SK, respectively). Ensure the information provided is correct, or the function may not work as expected. To create a secret, click **Create Secret** to go to the DEW console. For details, see **Creating a Secret**. Set the AK/SK in the **Secret key/value** tab. Enter **accessKeyId** and **secretAccessKey** in the **Key** column. To obtain the values, go to the DEW console, and choose **My Credentials** > **Access Keys**.	/

Configuring Deployment Management Settings

The following table describes the deployment management settings.

Table 1-24 Deployment management settings

Parameter	Description
Container Protocol	The network transmission protocol used by the container. Configure it according to the API definitions in your actual setup. If **Service Protocol** is set to **HTTP** or **HTTPS** in **Table 1-2**, **Container Protocol** can be set to **HTTP** or **HTTPS**. If **Service Protocol** is set to **WSS** or **WS** in **Table 1-2**, the container protocol is the same as the service protocol by default and is not displayed on the console.

Parameter	Description
Container Port	The port number that the image listens on. Requests are sent to the instance through this port. The port number must be the same as the port number in your image.
More Settings > Deployment Timeout (Minutes)	The wait time before a single service times out. This value includes both the deployment and startup time. Set this parameter properly. The system will cancel the deployment if it exceeds the time limit.
More Settings > Max Surge Replicas (%)	The maximum number of replicas that can be created above the target count during a rolling upgrade When percentages are used, the instance count is rounded up. Example: When **Max Surge Replicas (%)** is set to **1%** during a rolling upgrade of four instances, you can add one new instance at a time. When **Max Surge Replicas (%)** is set to **100%** during a rolling upgrade of two instances, you can add two new instances immediately. For details about rolling upgrade, see **Rolling Upgrades for Real-Time Service Deployment**.

Parameter	Description
More Settings > Max Unavailable Replicas (%)	The maximum number of replicas that can be unavailable relative to the target count during a rolling update When percentages are used, the instance count is rounded down. Example: • When **Max Unavailable Replicas (%)** is set to **1** during a rolling upgrade of four instances, four instances must remain available. An old instance can only be deleted after its replacement starts running. • When **Max Unavailable Replicas (%)** is set to **50** during a rolling upgrade of two instances, one old instance can be deleted immediately to release resources, but at least one instance must remain available. • If you deploy a service with **Max Unavailable Replicas (%)** at **20**, requiring five instances, and only one fails due to lack of resources while four start successfully, the deployment succeeds, and the service status becomes **Running**. If **Max Unavailable Replicas** equals the target replica count, there is a risk of downtime (**Min Available Replicas** = **Total Replicas** – **Max Unavailable Replicas**). Example: Setting **Max. Invalid Instances** to **100** allows the service stay running when an instance fails because of quick recovery. Yet, the service will not be able to perform predictions, potentially leading to interruptions. To ensure a hitless upgrade, set Max. Invalid Instances to 1%. (This requires extra resources. Make sure you have enough, or the update will fail.) For details about rolling upgrade, see **Rolling Upgrades for Real-Time Service Deployment**.

Configuring Advanced Settings

The following table describes the advanced settings.

Table 1-25 Advanced settings

Parameter	Description
Key Settings	Inference services support secret mounting. This uses encryption to protect sensitive data and securely mount it to containers. To protect your data, do not enter sensitive information in plaintext. • **Custom key**: Enter the key, key value, and mount path. • **DEW**: Data Encryption Workshop (DEW) is a comprehensive cloud data encryption service, including Key Management Service (KMS), Key Pair Service (KPS), and Dedicated Hardware Security Module (Dedicated HSM). To use DEW to configure the key, go to the DEW console to **create a key**, select the DEW key on the ModelArts console, and enter the mount path. Choose whether to select **Associate Image User Group ID**. The user group ID of the user who starts the image can be associated. After the association, the secret mounting security is improved.
Intelligent Routing Policy	If intelligent routing is enabled, set intelligent routing policies. The following policies are supported: • Round robin: Tasks are distributed sequentially to different nodes in order. This ensures an even distribution across the cluster and achieves load balancing. • Source IP hash: The system calculates hash values using client IP addresses to route requests from the same IP address to the same node. • Least connections: Requests are routed to the node with the fewest real-time connections. This method ensures that the load is distributed more evenly, improving service stability and response time. • Minimum time to first token: Requests are routed to the node with the lowest average time to first token. This method aims to minimize the wait time between receiving a request and starting its processing. This latency applies even when system resources are available immediately. • Overall loads: Requests are routed to the node with the lowest overall pressure, considering factors such as the number of connections, time to first token, and custom metrics. It directs new requests to less busy nodes to avoid resource waste and overloading. • SLO priority: Services are prioritized according to their assigned Service Level Objective (SLO) (0–3, where 0 is the highest). This ensures lower latency for higher-priority workloads. For details, see **Intelligent Routing Policy**.

Parameter	Description
Custom Metric Collection	If this function is enabled, you need to input the metric collection port details. Custom metrics will send data to AOM's Prometheus instances for viewing on the AOM console. To query custom metrics, see **Viewing Performance Metrics of a Real-Time Service on AOM** and **Viewing Performance Metrics of a Real-Time Service on ModelArts**. **Constraints** • ModelArts calls the HTTP API provided in the custom metric configuration every 10 seconds to obtain metric data. • The metric data text returned by the HTTP API provided in the custom metric configuration cannot exceed 32 KB. **Data Format of Custom Metrics** The format of custom metrics data must comply with the open metrics specifications. That is, the format of each metric must be: <*metric_name*>{<*tag_name*>=<*tag_value*>,...} <*sample_value*> [*timestamp_in_millisecond*] The following shows an example (the comment starts with #, which is optional): # HELP http_requests_total The total number of HTTP requests. # TYPE http_requests_total gauge html_http_requests_total{method="post",code="200"} 1656 1686660980680 html_http_requests_total{method="post",code="400"} 2 1686660980681
System Log Reporting	Displays only for NPU dedicated resource pools. When enabled, system logs are mounted to a fixed parallel file system for O&M engineers to analyze. System logs are stored for 30 days and will be automatically deleted after that. The mount path of system logs cannot be changed.

Confirming Settings

On the **Deploy Real-Time Service** > **Confirmation** page, confirm the configuration information and click **Confirm Deployment**.

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.

You can go to the real-time service list to check if the deployment is complete. Once the service status changes from **Deploying** to **Running**, the service is deployed.

NOTE

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

Follow-Up Operations

- For details about how to test a real-time service, compare the model effect in real time, and debug a real-time service using CloudShell, see **Testing a Real-Time Service**.

- For details about how to view real-time service details, see **Viewing Details About a Real-Time Service**.
- For details about how to modify, stop, and delete a real-time service, see **Managing the Lifecycle of a Real-Time Service Deployment**.

1.3 Testing a Real-Time Service

Accessing a Synchronous Real-Time Service

On the **ModelArts console**, choose **Model Inference** > **Real-Time Inference** in the left navigation pane. On the displayed real-time inference list, click **Call Service** in the **Operation** column of the target service to view the call information.

Alternatively, click the target service name to go to the service details page and obtain the call information in **Network Settings**.

Table 1-26 Call information for a synchronous real-time service

Type	URL Format	URL Example
Public API URL	https://{public-network-address}/v2/infer/{service-ID}	https://100.XX.XXX.XXX/v2/infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx

Using the Prediction Feature

After a service is deployed, verify its performance using the **Prediction** tab on the service details page. The service prediction tab offers the rest client. You can choose your service's request method and input the prediction path.

In the **Body** section, choose the appropriate data format (raw, binary, stream, or fromData).

- raw: Used to send raw text data, such as JSON, XML, or plaintext.
- binary: Used to upload binary files, such as images, audios, videos, or model files.
- stream: Block-based stream transmission is supported, which is applicable to real-time or continuous data input scenarios.
- fromData: Submit data in multipart/form-data format. Files can be uploaded and other text fields can be entered at the same time.

You can enter header information in **Headers**, for example, API key authentication information. Replace **{API Key}** with your own API key. Click **Predict** to send a prediction request.

Deleting the authorization key pair makes the system switch to IAM token authentication automatically.

For details about how to set the body and request header on the inference page, see **How Do I Fill in the Request Header and Request Body When a ModelArts Real-Time Service Is Running?**

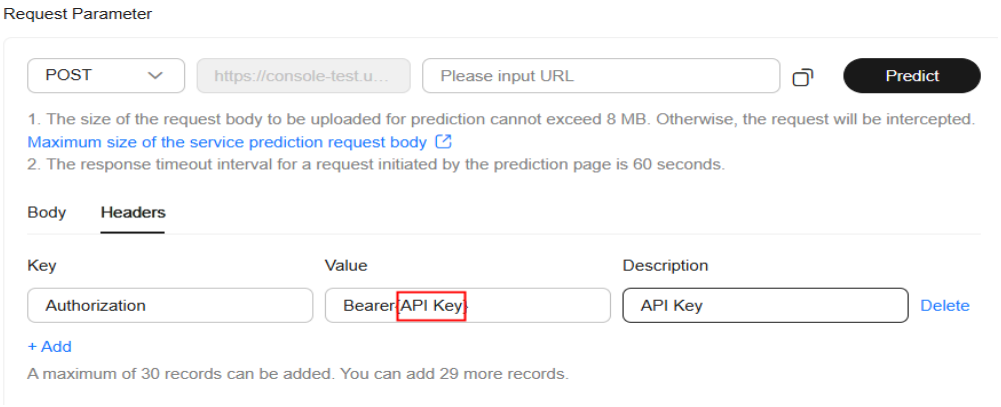
Example:

This guide demonstrates deploying the Qwen3-32B model using the Ascend-vLLM framework with one click. Here are the required parameters:

- Request method: POST
- Request path: https://***/v2/infer/***/v1/chat/completions
- Request body:

```
{
   "model": "qwen3_32b",
   "messages": [
     {"role": "system", "content": "You are a helpful assistant."},
     {"role": "user", "content": "Hello"}
   ]
}
```

Figure 1-14 Example headers



Using Cloud Shell to Debug a Real-Time Service Instance Container

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

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

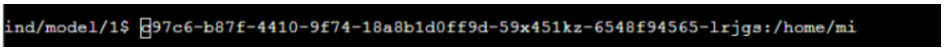
1. Log in to the ModelArts console. In the navigation pane on the left, choose **Model Inference** > **Real-Time Inference**.
2. Click the real-time service name or ID to access its details page.
3. In the **Cloud Shell** tab, select the deployment, instance, and Pod. If the connection status changes to **Connection Succeeded**, 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 Pod.

Figure 1-15 Cloud Shell

If you encounter a path display issue when logging in to Cloud Shell, press **Enter** to resolve the problem.

Figure 1-16 Path display issue



4. After logging in to the container, execute the necessary debugging commands in its terminal. Example:

   NOTE

   The following is for reference only. The actual log paths and service health check methods depend on your service configuration. Refer to your image settings and container startup commands for details.

   View logs:

   ```
   tail -f /var/log/app.log
   ```

   Check the service status:

   ```
   systemctl status app
   ```

   Run a custom script:

   ```
   ./debug_script.sh
   ```

5. After the debugging, exit the container:

   ```
   exit
   ```

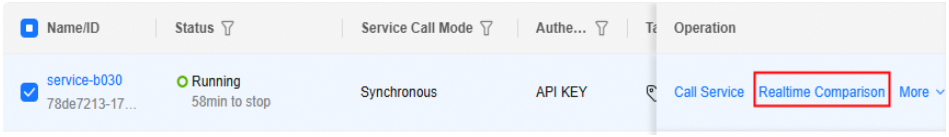
   After returning to the Cloud Shell terminal, you can view the debugging result or log file.

Live Comparison

Once a service is deployed, you can perform real-time inference comparisons with other services. This feature is currently only available for real-time text generation services. Live comparison can only be initiated when the real-time service status is **Running**, **Alarm**, or **Upgrading**.

1. On the **ModelArts console**, choose **Model Inference** > **Real-Time Inference**. On the displayed page, click **Live Comparison** in the **Operation** column on the right.

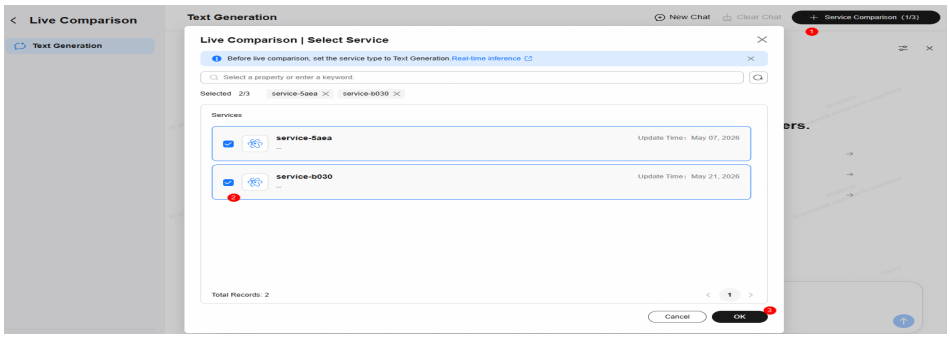
   Figure 1-17 Live comparison

2. Evaluate the Q&A service performance.

   To compare the performance of multiple models, click **Service Comparison** in the upper right corner. Select the target services and click **OK**. Currently, only text generation services are supported. Only deployed services can be selected for comparison.

Figure 1-18 Live comparison



Enter a prompt to compare the responses generated by the models.

For more information about live comparison, see **Live Comparison**.

1.4 Accessing a Real-time Service via Different Authentication Methods

1.4.1 Accessing a Real-Time Service with No Authentication

If a real-time service is in the **Running** state, it has been deployed successfully. This service provides a callable RESTful API. ModelArts allows you to test your model's performance instantly by accessing real-time services directly, with no authentication required.

Selecting **None** allows the client to call the API directly, which compromises security. This option is only suitable for temporary testing or internal network calls. Avoid using this mode on external networks. Use secure methods like **Accessing a Real-Time Service Through IAM Token-based Authentication** or **Accessing a Real-Time Service Through API Key Authentication** to verify access to real-time services.

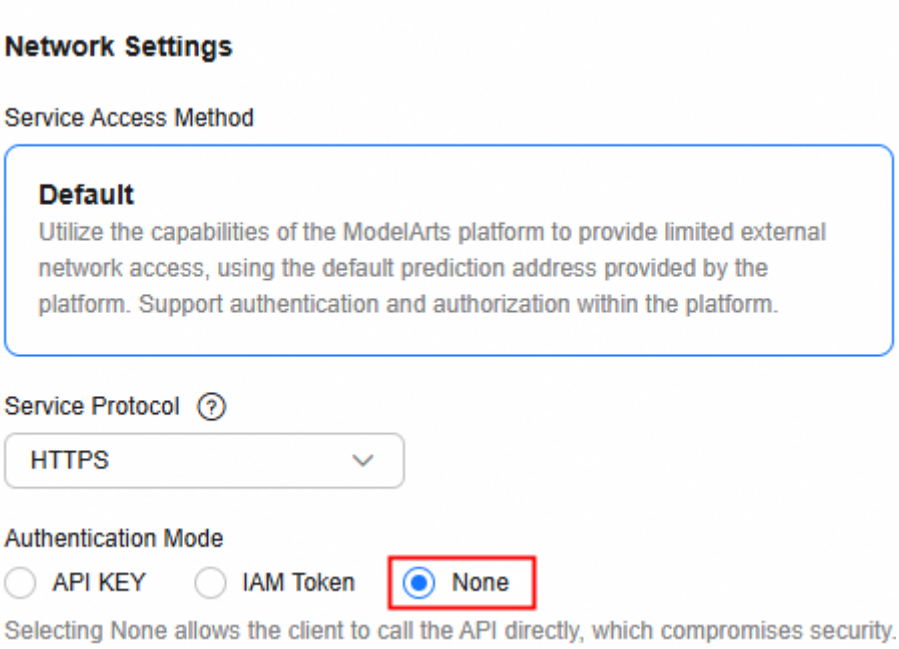
Before integrating into a production environment, you need to debug and test this API. You can use the following methods to send prediction requests to the real-time service:

- **Method 1: Use GUI-based Software for Prediction (Postman)**
- **Method 2: Send a Prediction Request via cURL**
- **Method 3: Use Python to Send a Prediction Request**
- **Method 4: Use Java to Send a Prediction Request**

Prerequisites

- **None** is selected as the authentication mode during **Procedure**.
- To prevent failed API calls, ensure that the number of concurrent requests, request body size, and request timeout interval do not exceed the limits set during deployment.

Figure 1-19 Configuring no authentication for a real-time service

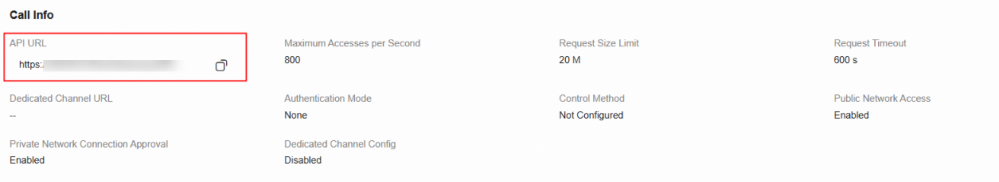


Obtaining the Prediction File Path and Real-Time Service URL

- The local path to the prediction 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**).
- The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Service** tab.

  API URL is the 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}***/v1/chat/completions**.

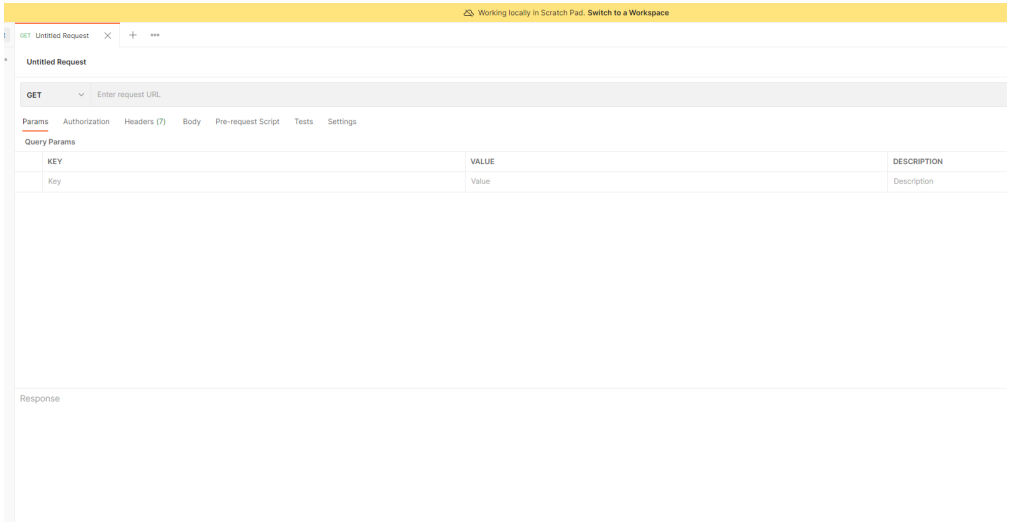
Figure 1-20 Obtaining the API URL of a real-time service



Method 1: Use GUI-based Software for Prediction (Postman)

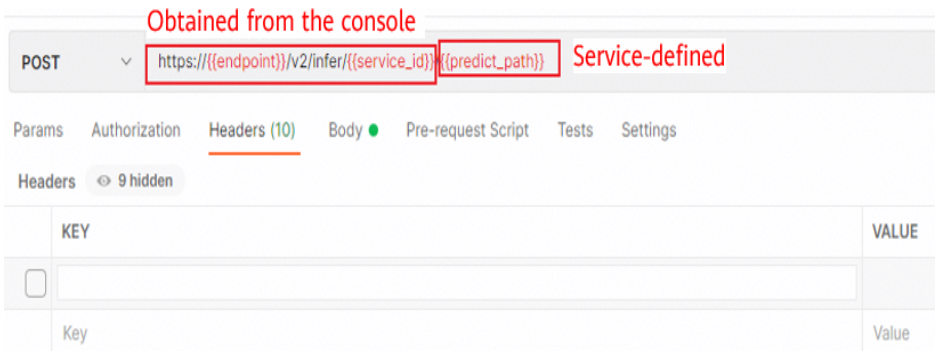
1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 8.11.1 is recommended.
2. Open Postman. **Figure 1-21** shows the Postman interface.

Figure 1-21 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.

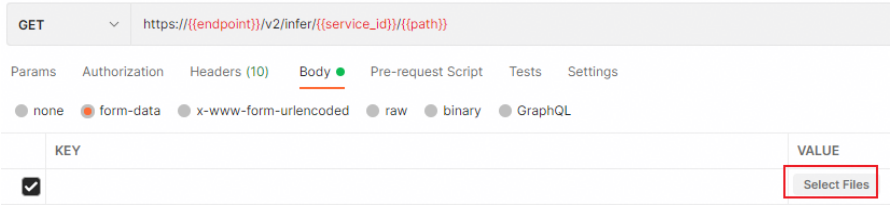
     Figure 1-22 Parameter settings

   - In the **Body** tab, file input and text input are available.
     - **File input**

       Select **form-data**. Set **KEY** to the input parameter of the model, 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 1-23**.

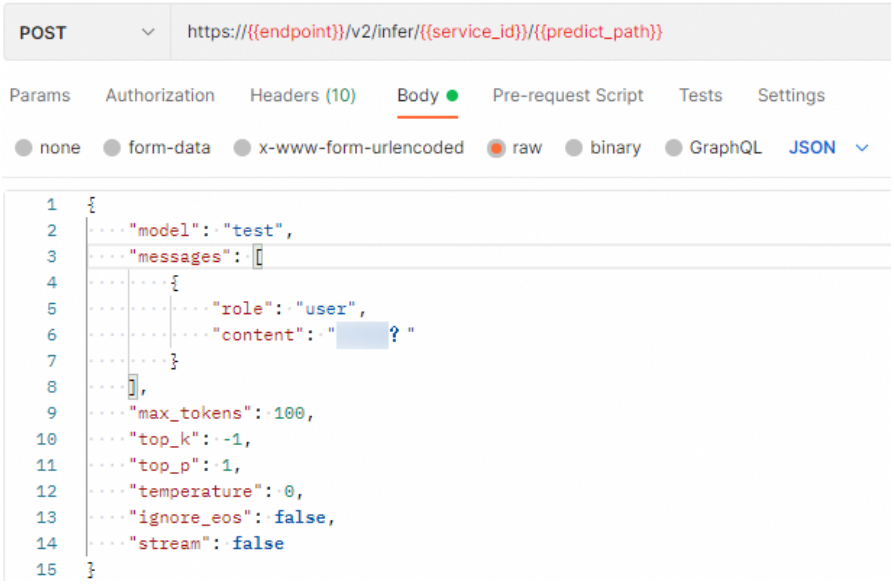
       Figure 1-23 Setting parameters in the **Body** tab

- **Text input**

  Select **raw** and **JSON (application/json)**, and enter the request body in the text box below. The format and contents of the request body depend on the model being used. The platform does not process the input data in any way.

  Figure 1-24 Text prediction request for a real-time service using no authentication

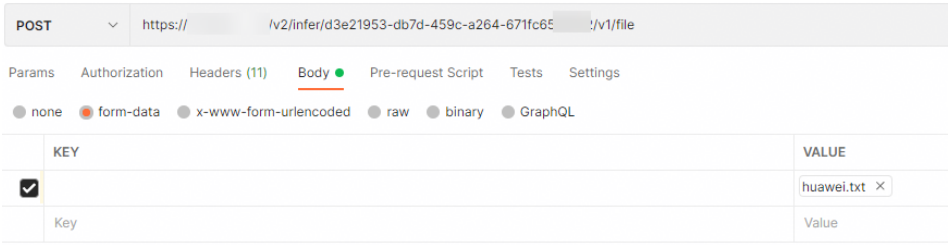
  

  Example request body:

  ```
  {
     "model": "test",
     "messages": [
        {
           "role": "user",
           "content": " Who are you?"
        }
     ],
     "max_tokens": 100,
     "top_k": -1,
     "top_p": 1,
     "temperature": 0,
     "ignore_eos": false,
     "stream": false
  }
  ```

4. After setting the parameters, click **send** to send the request. The result will be displayed in **Response**.
   - Prediction result using file input: **Figure 1-25** shows an example. The field values in the return result vary with the model.
   - **Figure 1-26** shows an example of prediction result for text input. The returned contents and format depend on the model being used.

Figure 1-25 File prediction result



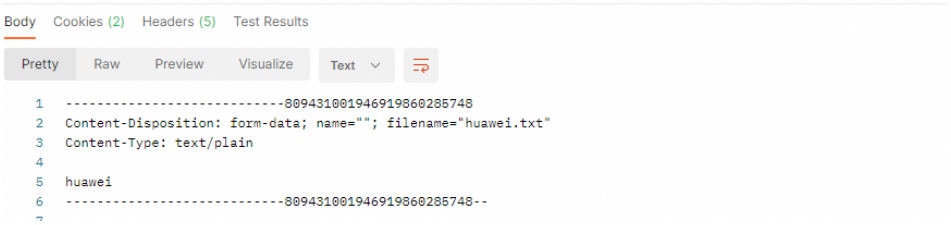
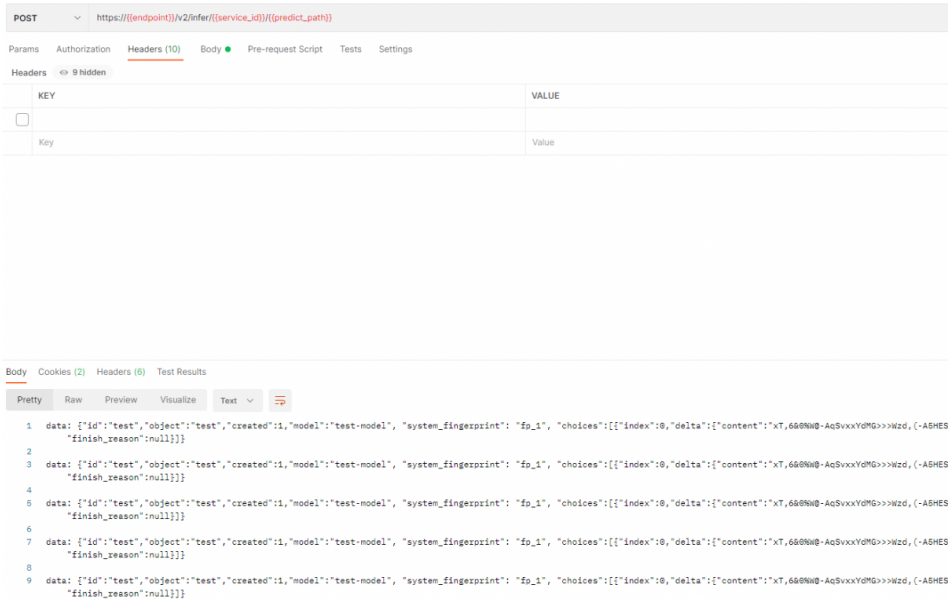


Figure 1-26 Text prediction result



Method 2: Send a Prediction Request via cURL

The **curl** command for sending prediction requests can be input as a file or text.

- File input
  ```
  curl -kv -F 'images=@Image path' -X POST Real-time service address
  ```
  - **-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 **@**.
  - **POST** is followed by the API URL of the real-time service.

  The following is an example of the curl command for prediction with file input:
  ```
  curl -kv -F 'images=@/home/data/test.png'  -X POST https://{{endpoint}}/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 'Content-type: application/
  json' -X POST https://{{endpoint}}/v1/infers/eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
  ```
  -d indicates the text input of the request body.

Method 3: Use Python to Send a Prediction 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 prediction.
   - **File input**
     ```
     # coding=utf-8

     import requests

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

         # Send request.
         files = {
             'images': open(file_path, 'rb')
         }
         resp = requests.post(url, 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 **Obtaining the Prediction File Path and Real-Time Service URL** is an example.
   - **Text input (JSON)**

     The following is an example of the request body for reading the local prediction file and performing Base64 encoding:
     ```
     # coding=utf-8

     import base64
     import requests

     if __name__ == '__main__':
         # Config url, token and file path
         url = "Real-time service URL"
     ```

```
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',
}
body = {
    'image': base64_data
}
resp = requests.post(url, headers=headers, json=body)

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

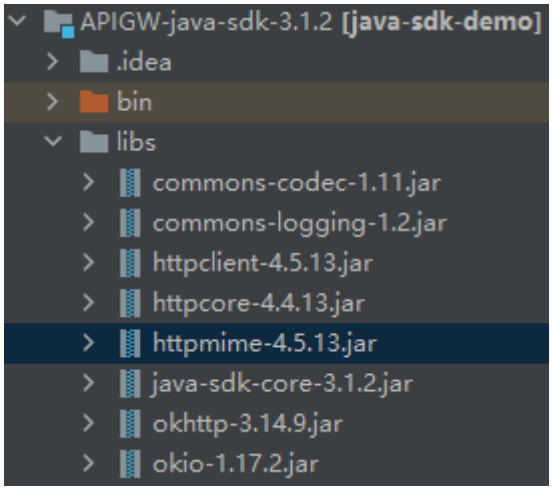
The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. The input parameter **images** obtained in **Obtaining the Prediction File Path and Real-Time Service URL** is an example. The value of **base64_data** in **body** is of the string type.

Method 4: Use Java to Send a Prediction 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 inference request input is in a file format, follow these steps to ensure the Java project includes the httpmime module as a dependency.
   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 1-27** 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 1-27 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 prediction.
   - **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 MyTokenFile {

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

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


            // 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 **Obtaining the Prediction File Path and Real-Time Service URL** is used as an example.

   - **Text input (JSON)**

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

```
// 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 = "Real-time service URL";
        String body = "{}";

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

            // Add header parameters
            httpPost.setHeader(HttpHeaders.CONTENT_TYPE, "application/json");

            // 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.

1.4.2 Accessing a Real-Time Service Through IAM Token-based Authentication

If a real-time service is in the **Running** state, it has been deployed successfully. This service provides a callable RESTful API. Directly using the API can create security risks by potentially letting unauthorized users view private information or execute actions. For secure API access, ModelArts uses token-based authentication for its real-time services.

During token-based authentication, the token is added to requests to get permissions for calling the API. ModelArts uses tokens for accessing real-time services. Tokens typically last 24 hours. Once expired, you must get a new one. This method suits fast development and testing due to its simplicity, though tokens expire quickly.

Test the real-time service's RESTful API before deploying it to production. Send an inference request using token-based authentication with these steps:

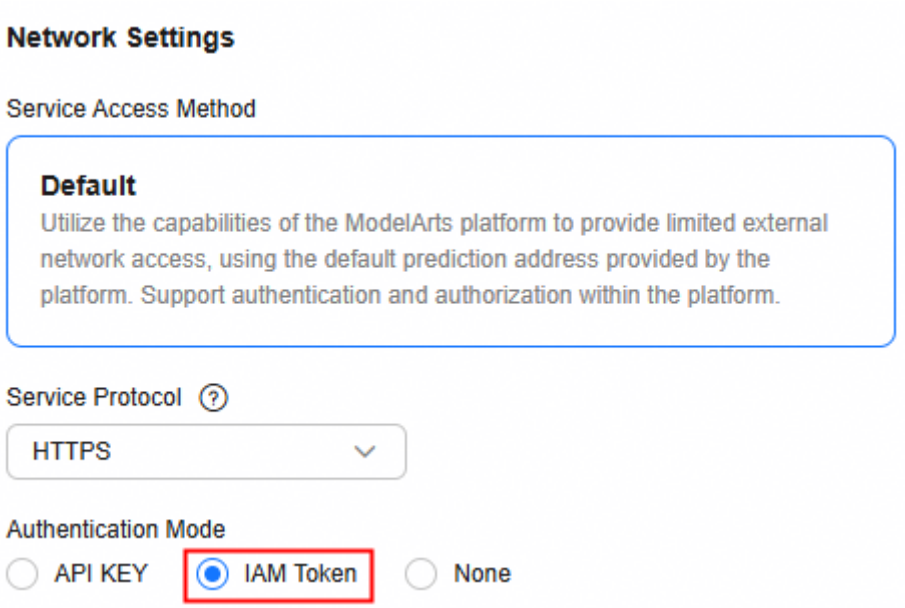
- **Method 1: Use GUI-based Software for Prediction (Postman)**. For Windows, Postman is recommended.

- **Method 2: Send a Prediction Request via cURL**. For Linux, cURL commands are recommended.
- **Method 3: Use Python to Send a Prediction Request**.
- **Method 4: Use Java to Send a Prediction Request**.

Prerequisites

- **IAM Token** is selected as the authentication mode during **Procedure**.
- To prevent failed API calls, ensure that the number of concurrent requests, request body size, and request timeout interval do not exceed the limits set during deployment.

Figure 1-28 Configuring IAM token authentication for a real-time service



Obtaining the User Token, Local Path to the Prediction File, and URL of the Real-Time Service

- **User Token**

  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.

- **Local Path to the Prediction File**

  The local path to the prediction 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**).

- **URL of the Real-Time Service**

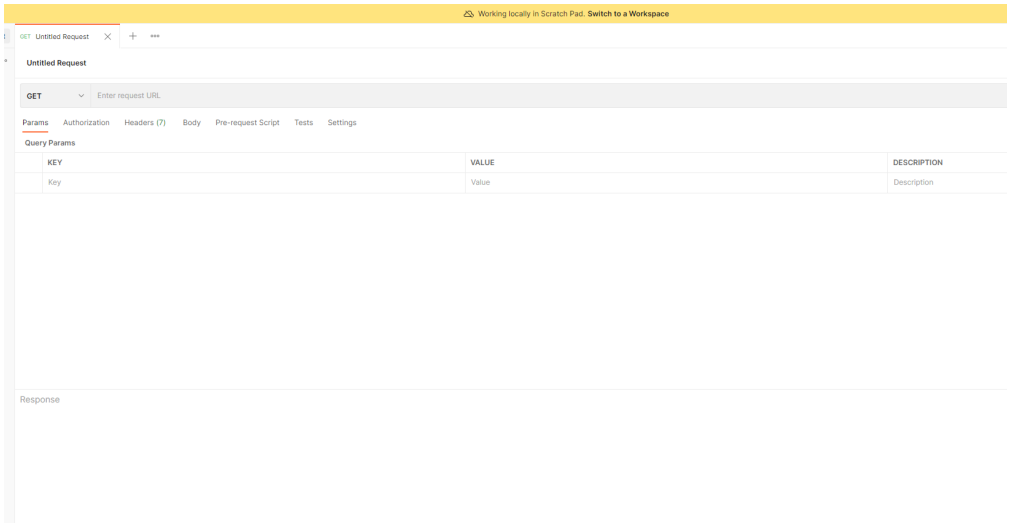
  The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Service** tab.

API URL is the 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}***/v1/chat/completions**.

Method 1: Use GUI-based Software for Prediction (Postman)

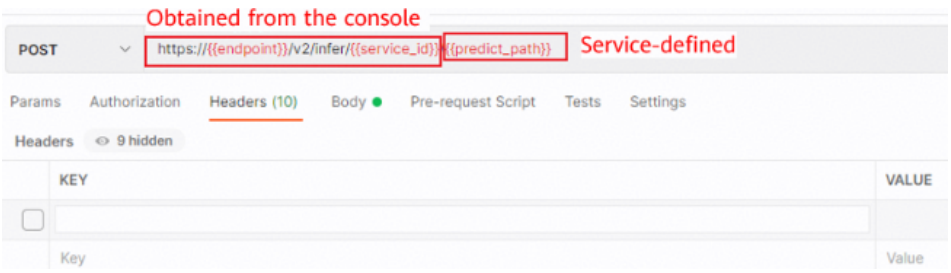
1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 8.11.1 is recommended.
2. Open Postman. **Figure 1-29** shows the Postman interface.

   Figure 1-29 Postman interface

3. Set parameters on Postman. The following uses image classification as an example.
   - Select a POST task and copy the obtained API URL to the POST text box. In the **Headers** tab, set **Key** to **X-Auth-Token** and **Value** to the user token.

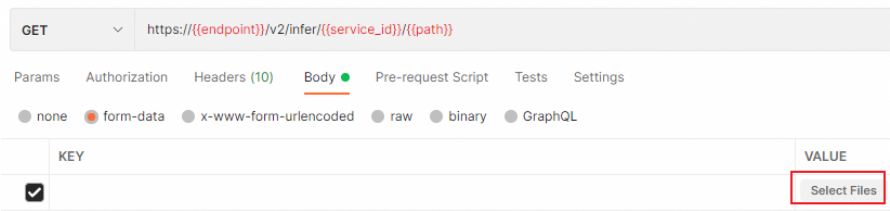
     Figure 1-30 Parameter settings

   - In the **Body** tab, file input and text input are available.
     - **File input**

       Select **form-data**. Set **KEY** to the input parameter of the model, 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 1-31**.

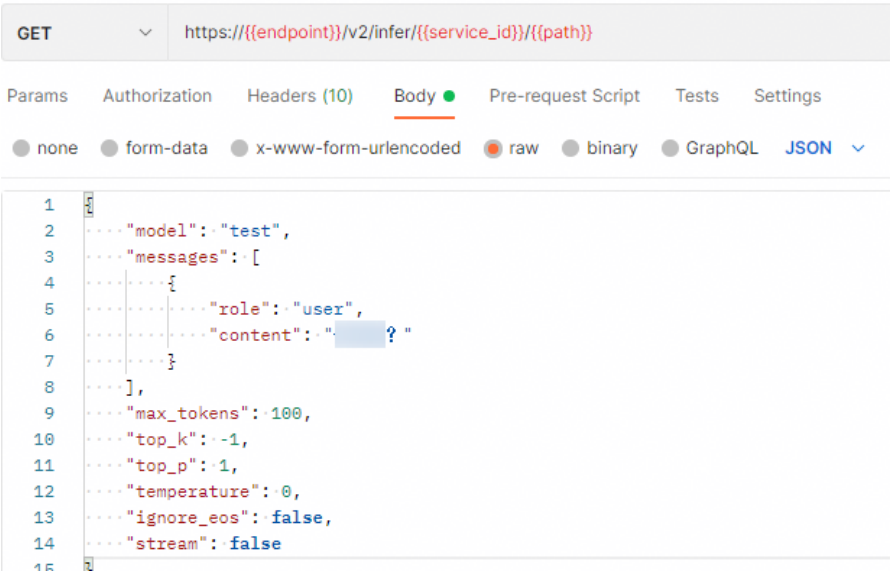
Figure 1-31 Setting parameters in the **Body** tab



- **Text input**

  Select **raw** and **JSON (application/json)**, and enter the request body in the text box below. The format and contents of the request body depend on the model being used. The platform does not process the input data in any way.

  Figure 1-32 Text prediction request for a real-time service using IAM authentication



  Example request body:

```
{
   "model": "test",
   "messages": [
      {
         "role": "user",
         "content": " Who are you?"
      }
   ],
   "max_tokens": 100,
   "top_k": -1,
   "top_p": 1,
   "temperature": 0,
   "ignore_eos": false,
   "stream": false
}
```

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

– Prediction result using file input: **Figure 1-33** shows an example. The field values in the return result vary with the model.

– **Figure 1-34** shows an example of prediction result for text input. The returned contents and format depend on the model being used.

Figure 1-33 File prediction result

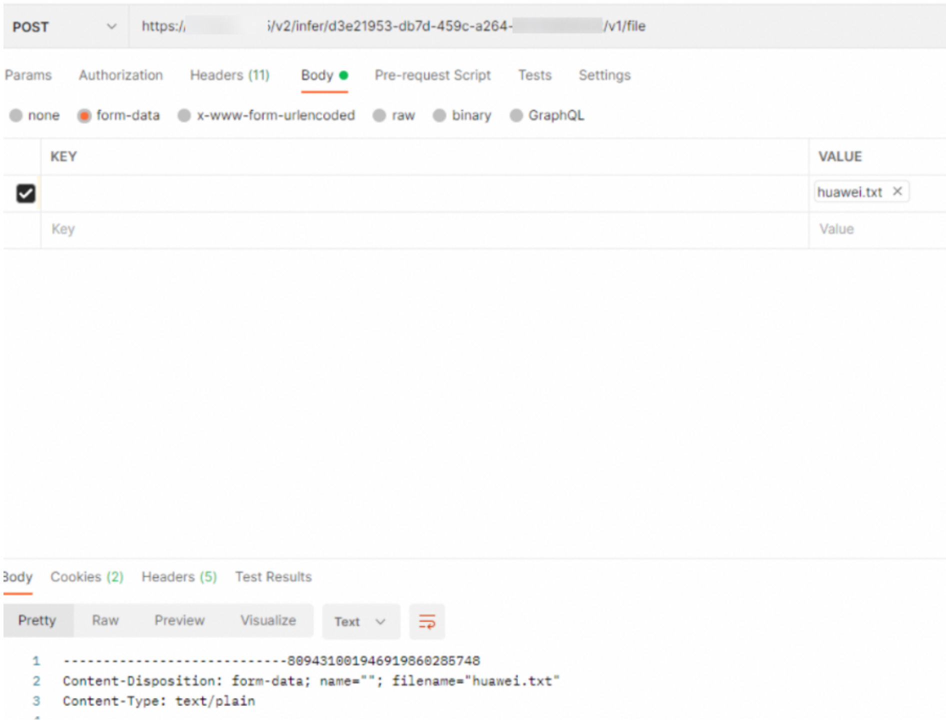
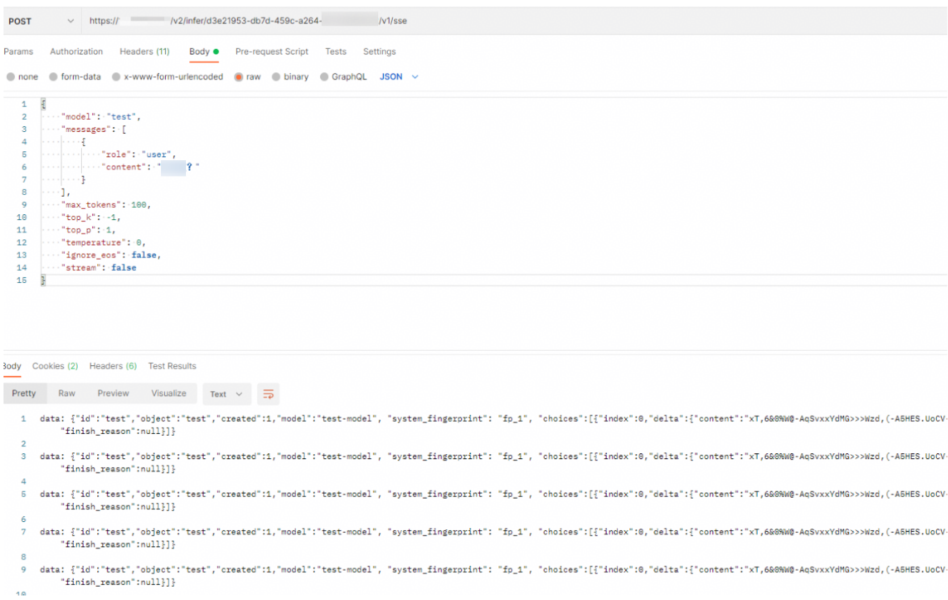


Figure 1-34 Text prediction result



Method 2: Send a Prediction Request via cURL

The **curl** command for sending prediction 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 prediction with file input:

  ```
  curl -kv -F 'images=@/home/data/test.png' -H 'X-Auth-Token:MIIS***80T9wHQ==' -X POST https://{{infer-endpoint}}/v2/infers/eb3e0c54-3dfa-4750-****-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:MII***80T9wHQ==' -H 'Content-type: application/json' -X POST https://{{infer-endpoint}}/v2/infers/eb3e0c54-3dfa-4750-****-95c45e5d3e83
  ```

  -d indicates the text input of the request body.

Method 3: Use Python to Send a Prediction Request

1. Download the Python SDK and configure the SDK in the development tool. For details, see **Integrating the Python SDK for API request signing**.
2. Create a request body for prediction.
   - **File input**

     ```
     # coding=utf-8

     import requests

     if __name__ == '__main__':
         # Config url, token and file path.
         url = "Real-time service URL"
         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 **Obtaining the User Token, Local Path to the Prediction File, and URL of the Real-Time Service** is an example.

- **Text input (JSON)**

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

```
# coding=utf-8

import base64
import requests

if __name__ == '__main__':
    # Config url, token and file path
    url = "Real-time service URL"
    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)
```

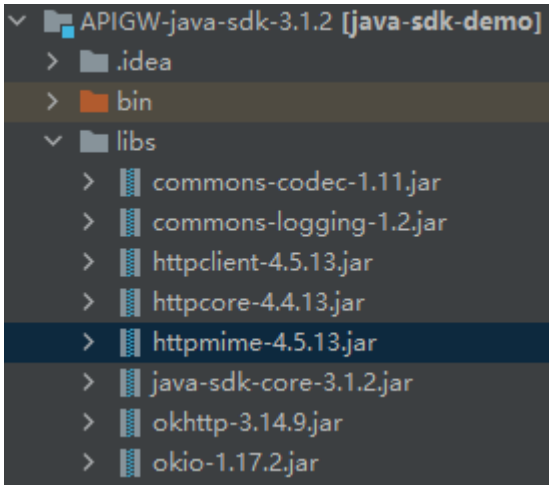
  The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. The input parameter **images** obtained in **Obtaining the User Token, Local Path to the Prediction File, and URL of the Real-Time Service** is an example. The value of **base64_data** in **body** is of the string type.

Method 4: Use Java to Send a Prediction 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 inference request input is in a file format, follow these steps to ensure the Java project includes the httpmime module as a dependency.
   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 1-35** 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 1-35 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 prediction.
   - **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 MyTokenFile {

    public static void main(String[] args) {
        // Config url, token and filePath
        String url = "Real-time service URL";
        String token = "User token";
        String filePath = "Local path to the prediction 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 **Obtaining the User Token, Local Path to the Prediction File, and URL of the Real-Time Service** is used as an example.

- **Text input (JSON)**

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

```
// 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 = "Real-time service URL";
        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.

1.4.3 Accessing a Real-Time Service Through API Key Authentication

If a real-time service is in the **Running** state, it has been deployed. This service provides a standard, callable RESTful API. When using real-time services provided by ModelArts, you need to make API calls for model predictions, typically requiring identity authentication to ensure security. However, traditional authentication methods may involve complex configurations and management, increasing development and maintenance challenges. How can we streamline the API call authentication process while maintaining security? ModelArts allows you to access real-time services through API key authentication.

To send a prediction request to a real-time service, first obtain your API key and add authentication information in the request header. The API key is used for authentication during API calls. Each user's API key serves as their unique identification credential, essential for accessing application APIs, and must be securely maintained. API key authentication works for basic security needs. It allows users with valid keys access the real-time service.

Test the real-time service's RESTful API before deploying it to production. Send an inference request using API key-based authentication with these steps:

- **Method 1: Use GUI-based Software for Prediction (Postman)**. For Windows, Postman is recommended.
- **Method 2: Send a Prediction Request via cURL**. For Linux, cURL commands are recommended.
- **Method 3: Use Python to Send a Prediction Request**.
- **Method 4: Use Java to Send a Prediction Request**.

Permissions

Table 1-27 Permissions required for API keys

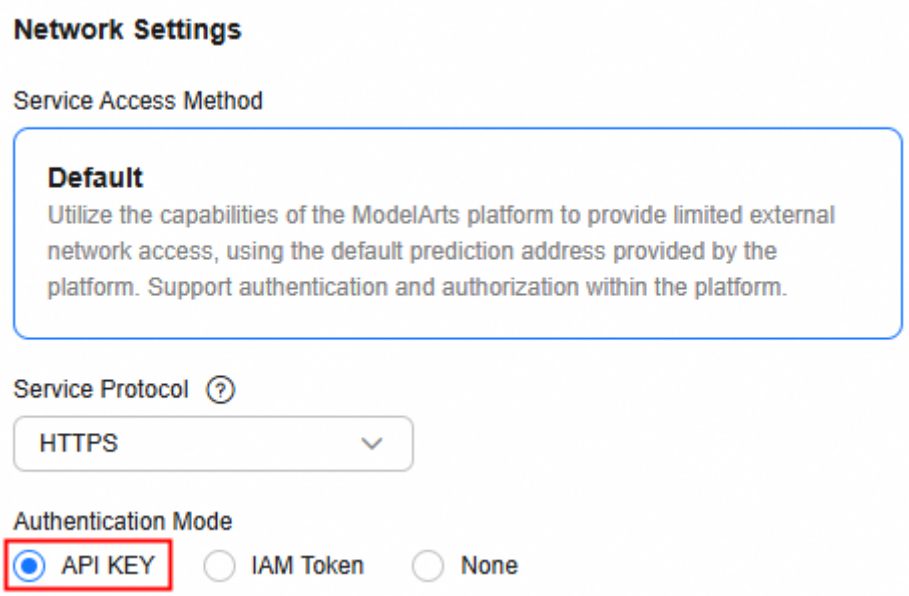
Permission	Action	Scenario
Obtaining model services	modelarts:service:list	Obtaining model services
Obtaining API keys	modelarts:apikey:list	Listing API keys
Unbinding an API key	modelarts:apikey:unbind	Unbinding an API key
Creating an API key	modelarts:apikey:create	Creating an API key
Binding an API key	modelarts:apikey:bind	Binding an API key

Permission	Action	Scenario
Deleting an API key	modelarts:apikey:delete	Deleting an API key

Prerequisites

- **API KEY** is selected as the authentication mode during **Configuring Service Information**.
- To prevent failed API calls, ensure that the number of concurrent requests, request body size, and request timeout interval do not exceed the limits set during deployment.

Figure 1-36 Configuring API key authentication for a real-time service



Managing API Keys

If you want to use API key authentication, create an API key on the authorization management page before deploying a real-time service and bind the API key to the real-time service. On the **Model Inference** > **Real-Time Inference** page, you can manage API keys, including creating, deleting, binding, and unbinding API keys.

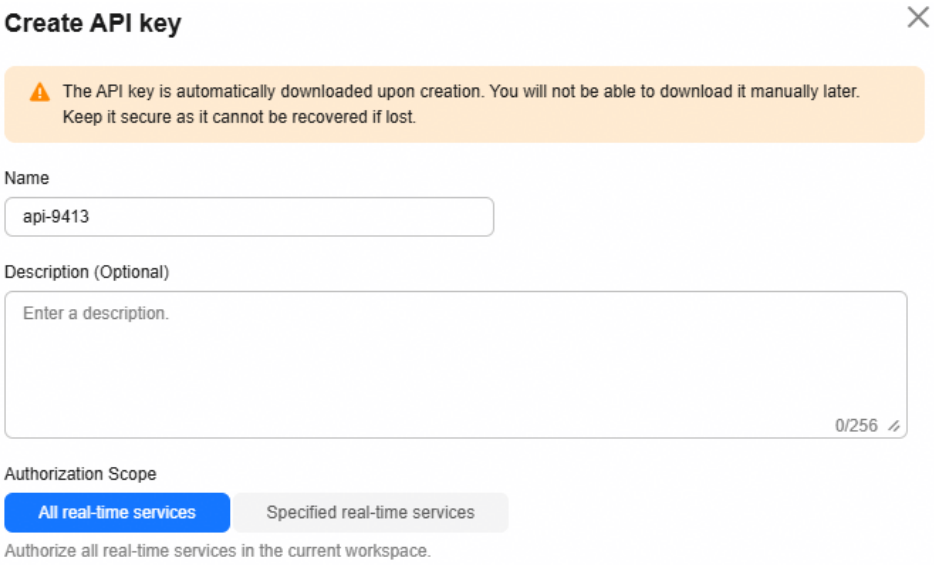
- Creating an API key
  a. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.
  b. Click **API Key Authorization Management** and click **Create**.
  c. In the displayed dialog box, enter the API key information based on **Table 1-28** and click **OK**.

The API key is automatically downloaded upon creation. You cannot download it manually. Keep it secure as it cannot be recovered if lost.

Table 1-28 API key parameters

Parameter	Description
Name	API key name, which must be unique.
Description (Optional)	Description of the API key.
Authorization Scope	Scope of the API key validity. • **All real-time services**: Authorize all real-time services in the current workspace. • **Specified real-time services**: Authorize only specified real-time services in the current workspace. An API key works only when **bound to a specific service** through the API authorization management page. Once bound, you can use this API key to access the real-time service.

Figure 1-37 Creating an API key



- Binding an API key to a service

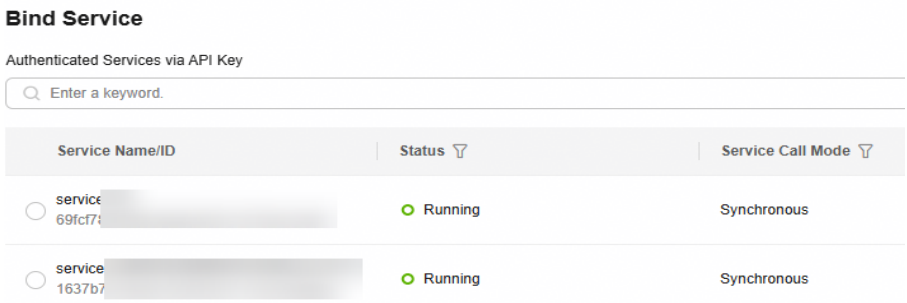
  If **Authorization Scope** is set to **Specified real-time services** during API key creation, the API key takes effect only after it is bound to a real-time service on the authorization management page. The service must be in **Running**, **Alarm**, **Stopping**, **Stopped**, or **Abnormal** state, with API key authentication mode.

  a. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.

b. Click **API Key Authorization Management**.
c. Locate the API key to be bound and click **Bind** in the **Operation** column.
d. In the displayed dialog box, select the service to be bound to the API key and click **OK**.

The API key is bound to the real-time service. You can use this key to call the service.

Figure 1-38 Binding an API key to a real-time service

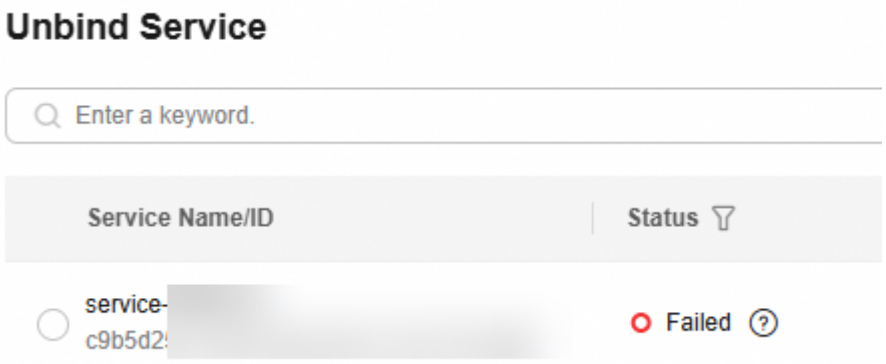


- Unbinding an API key from a service

  If the API key takes effect only on a real-time service, you can unbind the API key from the service.

  a. Log in to the ModelArts console. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.
  b. Click **API Key Authorization Management**.
  c. Click **Unbind** in the **Operation** column of the API key to be unbound from the real-time service.
  d. In the displayed dialog box, select the service to be unbound from the API key and click **OK**.

  Figure 1-39 Unbinding an API key from a service



- Deleting an API key

  You can delete an API key if it is no longer needed. The API key cannot be deleted if it is active for specific real-time services and bound to any of them.

  a. Log in to the ModelArts console. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.

b. Click **API Key Authorization Management**.
c. Click **Delete** in the **Operation** column of the API key to be deleted.
d. In the displayed dialog box, enter **DELETE** and click **OK**.

Obtaining Service Call Information

Before calling a service, **create an API key** and **bind the API key to the target real-time service**. Obtain the local path of the prediction file, the API URL of the real-time service, and the input parameters of the real-time service.

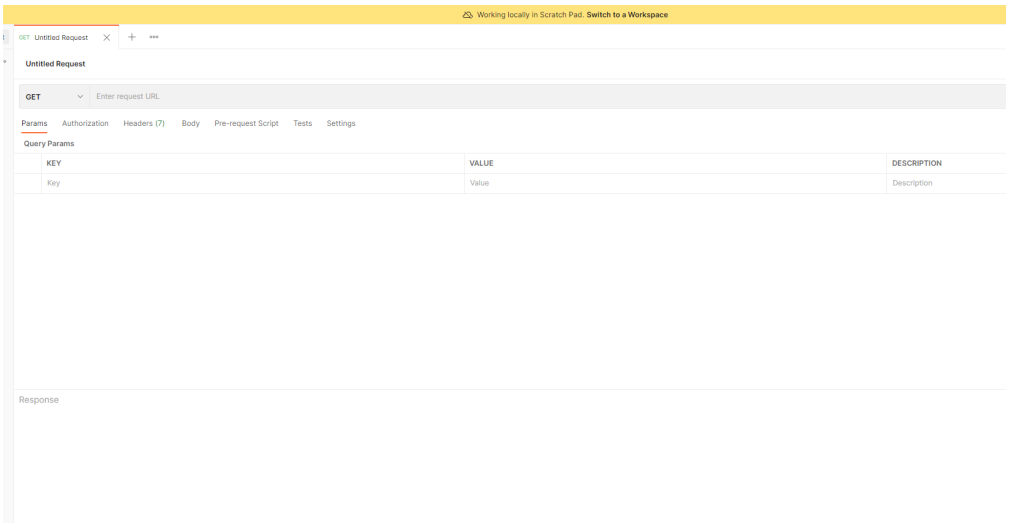
- Obtain the API key content: Open the CSV file automatically downloaded when you create the API key. Find the **api_key** field to get the API key content.
- The local path to the prediction file can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).
- The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Basic Information** tab.

  API URL is the 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}***/v1/chat/completions**.

Method 1: Use GUI-based Software for Prediction (Postman)

1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 8.11.1 is recommended.
2. Open Postman. **Figure 1-40** shows the Postman interface.

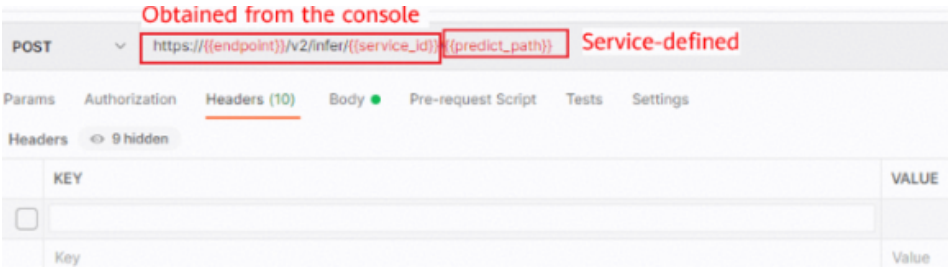
   Figure 1-40 Postman interface



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. In the **Headers** tab, set **KEY** to **Authorization** and **VALUE** to **Bearer** *<API-key-content>*.

  Figure 1-41 Parameter settings

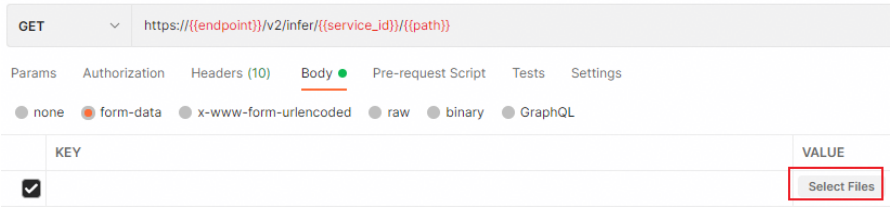
  

- In the **Body** tab, file input and text input are available.

  - **File input**

    Select **form-data**. Set **KEY** to the input parameter of the model, 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 1-42**.

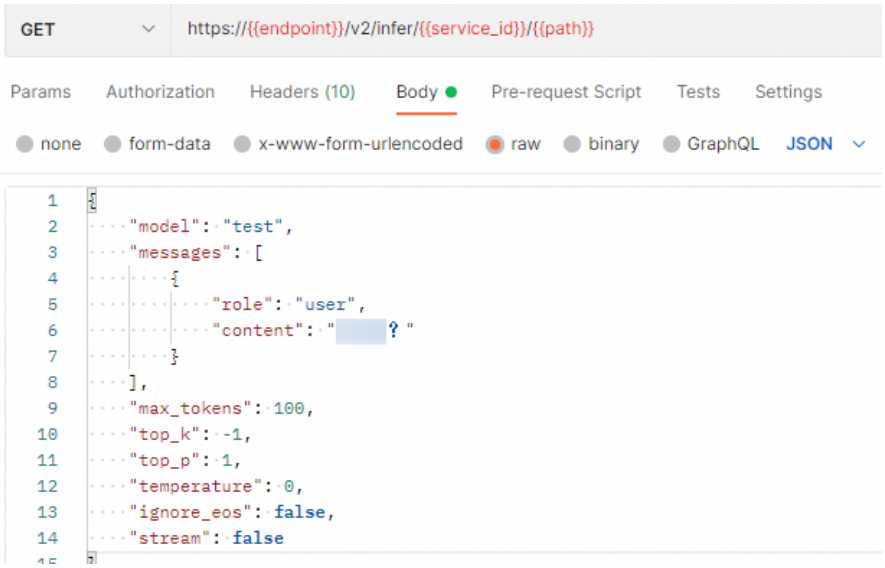
    Figure 1-42 Setting parameters in the **Body** tab

  - **Text input**

    Select **raw** and **JSON (application/json)**, and enter the request body in the text box below. The format and contents of the request body depend on the model being used. The platform does not process the input data in any way.

Figure 1-43 Text prediction request for a real-time service using IAM authentication



Example request body:

```
{
   "model": "test",
   "messages": [
      {
         "role": "user",
         "content": " Who are you?"
      }
   ],
   "max_tokens": 100,
   "top_k": -1,
   "top_p": 1,
   "temperature": 0,
   "ignore_eos": false,
   "stream": false
}
```

4. After setting the parameters, click **send** to send the request. The result will be displayed in **Response**.
   - Prediction result using file input: **Figure 1-44** shows an example. The field values in the return result vary with the model.
   - **Figure 1-45** shows an example of prediction result for text input. The returned contents and format depend on the model being used.

Figure 1-44 File prediction result

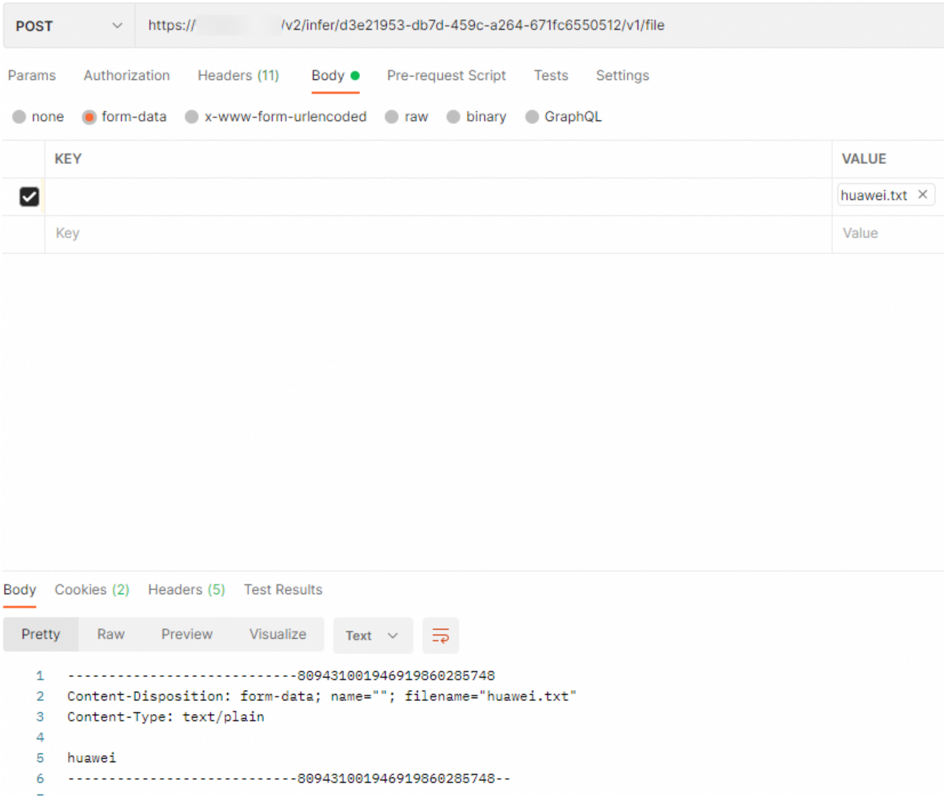
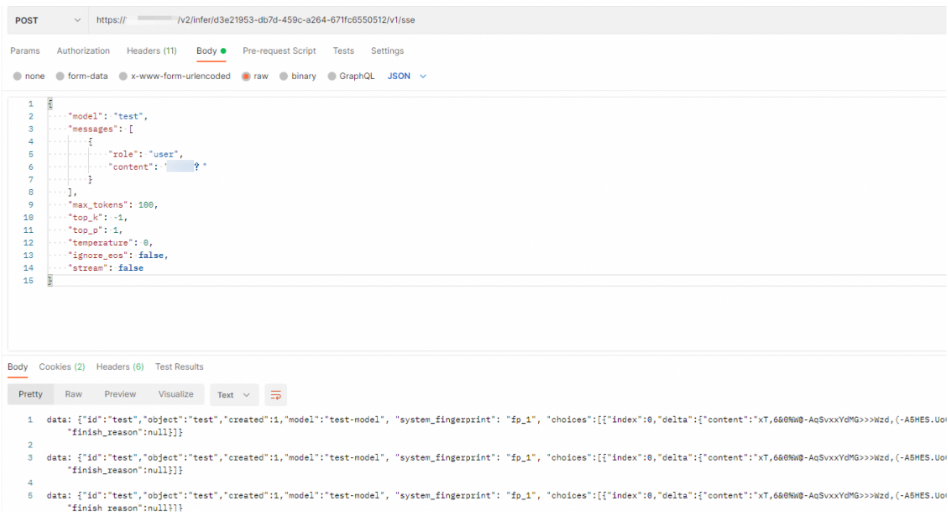


Figure 1-45 Text prediction result



Method 2: Send a Prediction Request via cURL

The **curl** command for sending prediction requests can be input as a file or text.

- File input
  curl -kv -F 'images=@ *Image path*' **-H 'Authorization:Bearer** *API key content*' **-X POST** *Real-time service address*

- **-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** specifies the post command's headers. It includes the **Authorization** header with a fixed key value. *API key content* contains the API key details.
- **POST** is followed by the API URL of the real-time service.

The following is an example of the curl command for prediction with file input:

```
curl -kv -F 'images=@/home/data/test.png' -H 'Authorization:Bearer 4**************w' -X POST https://{{infer-endpoint}}/v2/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 'Authorization:Bearer 4**************w' -H 'Content-type: application/json' -X POST https://{{infer-endpoint}}/v2/infers/eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
```

-d indicates the text input of the request body.

Method 3: Use Python to Send a Prediction 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 prediction.
   - **File input**

     ```
     # coding=utf-8

     import requests

     if __name__ == '__main__':
         # Config url, token and file path.
         url = "Real-time service URL"
         api_key= "API key content"
         file_path = "Local path to the inference file"

         # Send request.
         headers = {
             'Authorization': 'Bearer ' + api_key
         }
         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 **Obtaining Service Call Information** is an example.
   - **Text input (JSON)**

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

     ```
     # coding=utf-8

     import base64
     ```



```
import requests

if __name__ == '__main__':
    # Config url, token and file path
    url = "Real-time service URL"
    api_key= "API key content"
    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',
        'Authorization': 'Bearer ' + api_key
    }
    body = {
        'image': base64_data
    }
    resp = requests.post(url, headers=headers, json=body)

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

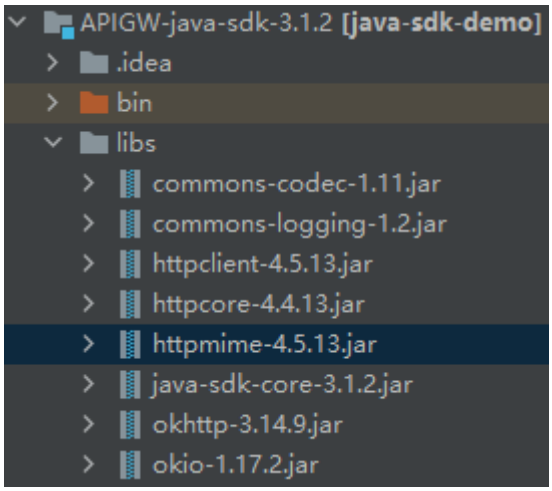
The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. The input parameter **images** obtained in **Obtaining Service Call Information** is an example. The value of **base64_data** in **body** is of the string type.

Method 4: Use Java to Send a Prediction 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 inference request input is in a file format, follow these steps to ensure the Java project includes the httpmime module as a dependency.
   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 1-46** 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 1-46 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 prediction.
   - **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 MyTokenFile {

    public static void main(String[] args) {
        // Config url, token and filePath
        String url = "Real-time service URL";
        String apiKey = "API key content";
        String filePath = "Local path to the prediction file";

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

            // Add header parameters
            httpPost.setHeader("Authorization", "Bearer " + api_key);

            // 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 **Obtaining Service Call Information** is used as an example.

- **Text input (JSON)**

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

```
// 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 = "Real-time service URL";
        String apiKey = "API key content";
        String body = "{}";

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

            // Add header parameters
            httpPost.setHeader(HttpHeaders.CONTENT_TYPE, "application/json");
            httpPost.setHeader("Authorization", "Bearer " + api_key);

            // 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.

1.5 Accessing a Real-Time Service Through Different Channels

1.5.1 Accessing a Real-Time Service Through a Public Network

Context

ModelArts inference supports access to real-time services through the public network, supporting both HTTPS and WebSocket protocols. A standard, callable RESTful API is provided after deployment of a real-time service.

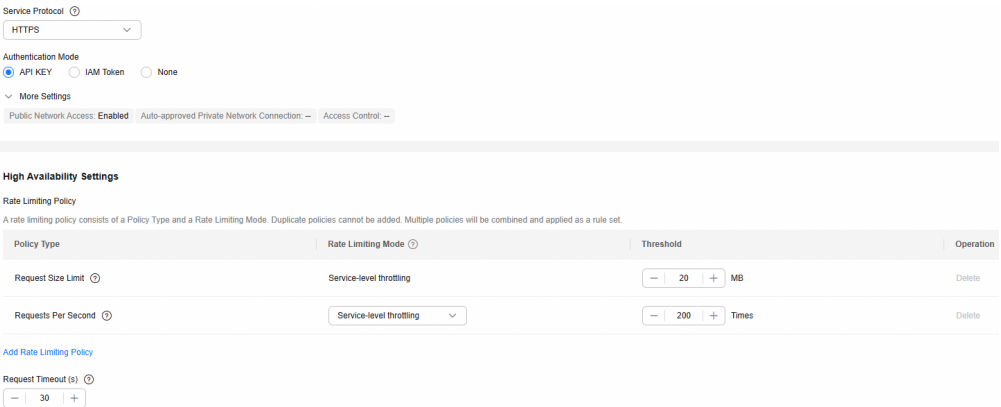
Test the real-time service's RESTful API before deploying it to production. This section describes how to send a prediction request to a real-time service using API key authentication.

- **Method 1: Use GUI-based Software for Prediction (Postman)**. For Windows, Postman is recommended.
- **Method 2: Send a Prediction Request via cURL**. For Linux, cURL commands are recommended.
- **Method 3: Use Python to Send a Prediction Request**.
- **Method 4: Use Java to Send a Prediction Request**.

Prerequisites

- To access a service from a public network, select **Public Network Access** during **Procedure**.
- This section uses API key authentication as an example. During **Procedure**, select API key authentication. Before calling a service, **create an API key** and **bind the API key to the real-time service to be accessed**.
- To prevent failed API calls, ensure that the number of concurrent requests, request body size, and request timeout interval do not exceed the limits set during deployment.

Figure 1-47 Configuring API key authentication for a real-time service



Obtaining Service Call Information

Before calling a service, **create an API key** and bind the API key to the target real-time service. Obtain the local path of the prediction file, the API URL of the real-time service, and the input parameters of the real-time service.

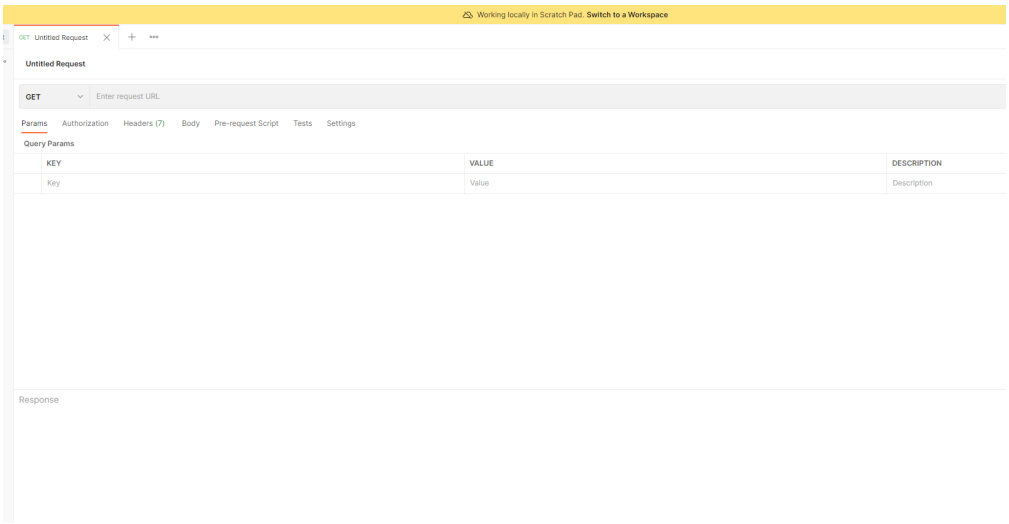
- Obtain the API key content: Open the CSV file automatically downloaded when you create the API key. Find the **api_key** field to get the API key content.
- The local path to the prediction file can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).
- The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Basic Information** tab.

  API URL is the 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}*/**v1/chat/completions**.

Method 1: Use GUI-based Software for Prediction (Postman)

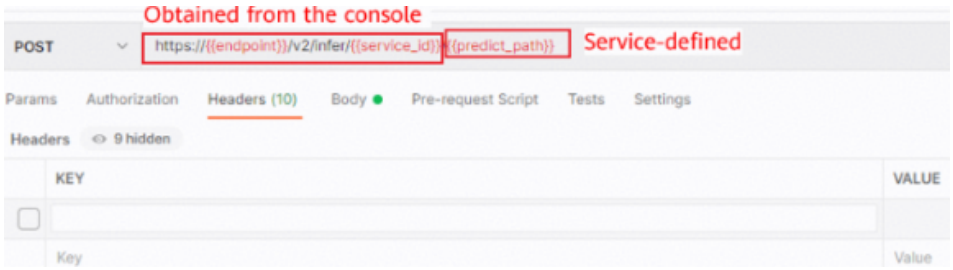
1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 8.11.1 is recommended.
2. Open Postman. **Figure 1-48** shows the Postman interface.

   Figure 1-48 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. In the **Headers** tab, set **KEY** to **Authorization** and **VALUE** to **Bearer** *<API-key-content>*.

Figure 1-49 Parameter settings

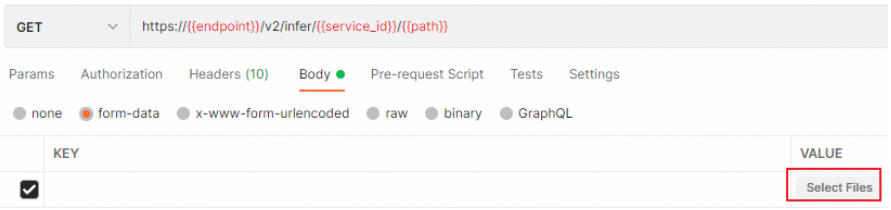


– In the **Body** tab, file input and text input are available.

  - **File input**

    Select **form-data**. Set **KEY** to the input parameter of the model, 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 1-50**.

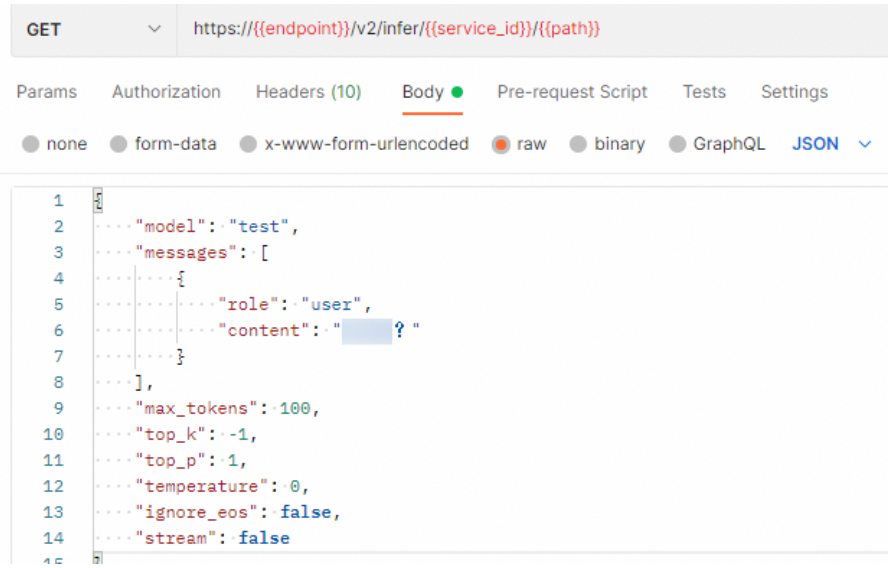
    Figure 1-50 Setting parameters in the **Body** tab



  - **Text input**

    Select **raw** and **JSON (application/json)**, and enter the request body in the text box below. The format and contents of the request body depend on the model being used. The platform does not process the input data in any way.

    Figure 1-51 Text prediction request for a real-time service using IAM authentication



Example request body:

```
{
   "model": "test",
   "messages": [
      {
         "role": "user",
         "content": " Who are you?"
      }
   ],
   "max_tokens": 100,
   "top_k": -1,
   "top_p": 1,
   "temperature": 0,
   "ignore_eos": false,
   "stream": false
}
```

4. After setting the parameters, click **send** to send the request. The result will be displayed in **Response**.
   - Prediction result using file input: **Figure 1-52** shows an example. The field values in the return result vary with the model.
   - **Figure 1-53** shows an example of prediction result for text input. The returned contents and format depend on the model being used.

Figure 1-52 File prediction result

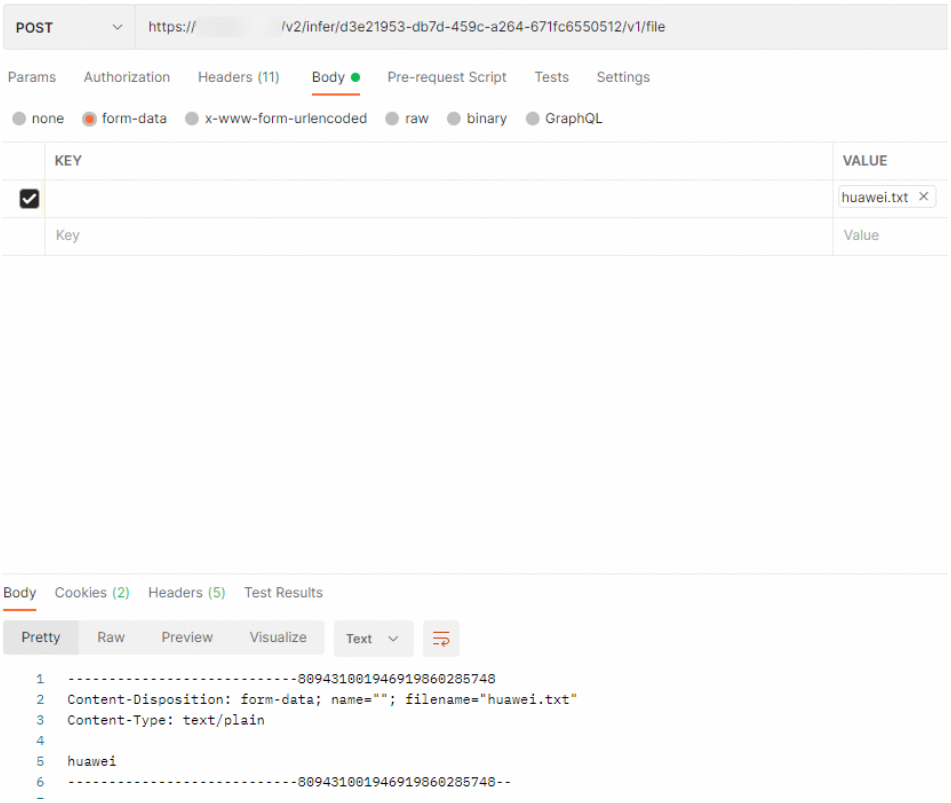
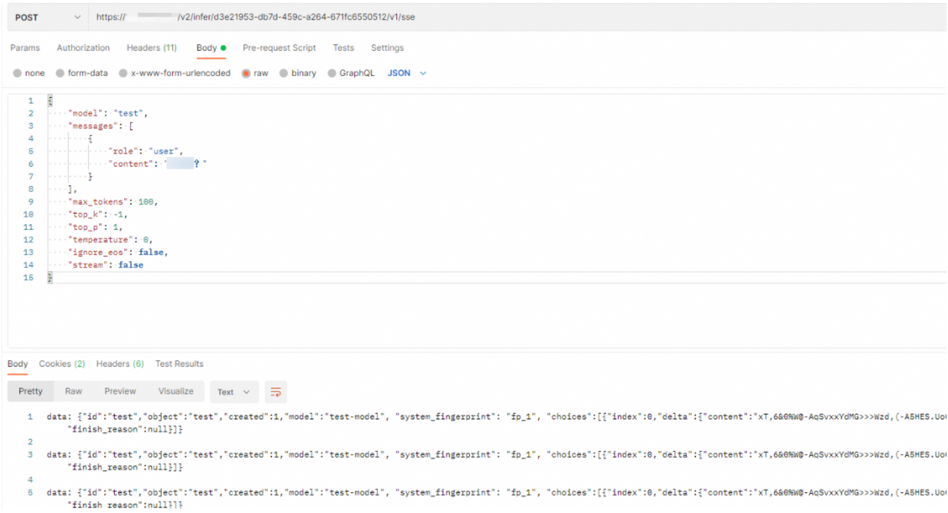


Figure 1-53 Text prediction result



Method 2: Send a Prediction Request via cURL

The **curl** command for sending prediction requests can be input as a file or text.

- File input
  curl -kv -F 'images=@ *Image path*' **-H 'Authorization:Bearer** *API key content*' **-X POST** *Real-time service address*
  – **-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** specifies the post command's headers. It includes the **Authorization** header with a fixed key value. *API key content* contains the API key details.
  – **POST** is followed by the API URL of the real-time service.

  The following is an example of the curl command for prediction with file input:

```
curl -kv -F 'images=@/home/data/test.png' -H 'Authorization:Bearer 4*************w' -X POST https://
{{infer-endpoint}}/v2/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 'Authorization:Bearer
4*************w' -H 'Content-type: application/json' -X POST https://{{infer-endpoint}}/v2/infers/
eb3e0c54-3dfa-4750-af0c-95c45e5d3e83
```

  -d indicates the text input of the request body.

Method 3: Use Python to Send a Prediction 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 prediction.
   – **File input**

```
# coding=utf-8
```

```
import requests

if __name__ == '__main__':
    # Config url, token and file path.
    url = "Real-time service URL"
    api_key= "API key content"
    file_path = "Local path to the inference file"

    # Send request.
    headers = {
        'Authorization': 'Bearer ' + api_key
    }
    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 **Obtaining Service Call Information** is an example.

- **Text input (JSON)**

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

```
# coding=utf-8

import base64
import requests

if __name__ == '__main__':
    # Config url, token and file path
    url = "Real-time service URL"
    api_key= "API key content"
    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',
        'Authorization': 'Bearer ' + api_key
    }
    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 **Obtaining Service Call Information** is an example. The value of **base64_data** in **body** is of the string type.

Method 4: Use Java to Send a Prediction Request

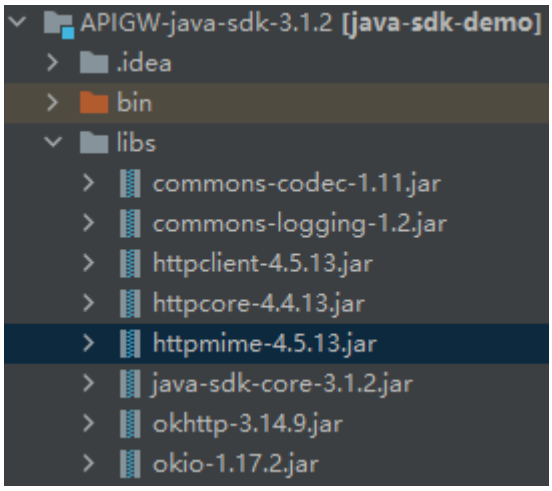
1. Download the Java SDK and configure it in the development tool. For details, see **Integrating the Java SDK for API request signing**.

2. (Optional) If the inference request input is in a file format, follow these steps to ensure the Java project includes the httpmime module as a dependency.

   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 1-54** 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 1-54 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 prediction.

   – **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 MyTokenFile {
```



```
    public static void main(String[] args) {
        // Config url, token and filePath
        String url = "Real-time service URL";
        String apiKey = "API key content";
        String filePath = "Local path to the prediction file";

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

            // Add header parameters
            httpPost.setHeader("Authorization", "Bearer " + api_key);

            // 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 **Obtaining Service Call Information** is used as an example.

- **Text input (JSON)**

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

```
// 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 = "Real-time service URL";
        String apiKey = "API key content";
        String body = "{}";

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

            // Add header parameters
            httpPost.setHeader(HttpHeaders.CONTENT_TYPE, "application/json");
            httpPost.setHeader("Authorization", "Bearer " + api_key);

            // 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.

Helpful Links

The authentication modes below are available for public network access to real-time services. For details about API calls, see:

- **Accessing a Real-Time Service Through IAM Token-based Authentication**
- **Accessing a Real-Time Service with No Authentication**
- **Accessing a Real-Time Service Through API Key Authentication**

1.5.2 Accessing a Real-Time Service Through a Private Network

Context

During enterprise-grade application development, when developers need to access the real-time inference service provided by ModelArts from the enterprise intranet, access obstacles may occur due to network isolation, affecting development efficiency and security. ModelArts supports accessing real-time services via private networks.

Private network access is suitable for scenarios requiring high security and low latency. For private network connection to real-time services, creating a VPCEP within one's own VPC or resource pool VPC offers a convenient, secure, and private channel to connect to the real-time inference service. ModelArts facilitates private network connectivity; by initiating a private network connection request, it automatically creates a VPCEP, establishing a direct private network connection between the VPC and the real-time inference service, thereby enabling efficient and secure access to the real-time inference service over the private network.

Permissions

When using the private network connection function, you must have the following permissions:

Table 1-29 Permissions required for private network connection

Permission	Action	Scenario
Obtaining details about a workspace	modelarts:workspace:get	Accessing workspaces
Obtaining model services	modelarts:service:list	Obtaining services
Obtaining dedicated resource pools	modelarts:pool:list	Obtaining resource pools
Obtaining VPCs	vpc:vpcs:list	Private network connection page and private network connection request creation
Obtaining subnets or subnet details	vpc:subnets:get	Private network connection page and private network connection request creation
Obtaining private network connections	modelarts:intranetConnection:list	Querying the private network connection list
Modifying private network connections	modelarts:intranetConnection:update	Modifying private network connections
Creating private network connections	modelarts:intranetConnection:create	Creating private network connections
Deleting private network connections	modelarts:intranetConnection:delete	Deleting private network connections

You also need to grant the agency permissions below to ModelArts. For details about how to add ModelArts authorization, see **Configuring Agency Authorization for ModelArts with One Click**.

Table 1-30 Permissions granted to ModelArts for private network connection

Permission	Action	Scenario
Creating endpoints	vpcep:endpoints:create	Creating and changing private network connections
Deleting endpoints	vpcep:endpoints:delete	Deleting private network connections
Querying endpoint details	vpcep:endpoints:get	Creating and changing private network connections
Querying endpoints	vpcep:endpoints:list	Creating and changing private network connections

Constraints

- In the user network scenario, private network connection is VPC-based. You can create only one private network connection for a service in the same VPC. You can only select your own VPC network to access the private network.
- In the resource pool network scenario, private network connection is resource pool network-based. You can create only one private network connection for a resource pool in the same resource pool network.

Prerequisites

- The real-time service is running properly.
- You have obtained the real-time service ID on the service details page by clicking the service name in the inference service list.

Figure 1-55 Obtaining the real-time service ID

- To enable the private network connection between your VPC and inference service, create a VPC and subnet first. For details, see **Creating a VPC with a Subnet**.
- To enable the private network connection between your resource pool and inference service, create a dedicated resource pool first. For details, see **Creating a Dedicated Resource Pool**.

Creating a Private Network Connection

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.
2. Click **Private Network Connections**.
3. Click the **My Requests** tab. Then, click **Create**.
4. Set parameters and click **OK**.

   Check the request status in the **My Requests** tab. If the request status is **Connection succeeded**, the request is approved. Record the **Access Address**. If **Auto-approved Private Network Connection** is selected during **service information configuration**, the requests from third-party users will be approved automatically.

   When the request is in the **Abnormal** state and the reason is "Connection failed. Try again", click **Retry** in the **Operation** column to create the private network connection again.

   After the private network connection request is approved, if the request scenario is user network, see **Accessing a Real-Time Service Using Your VPC**. If the request scenario is resource pool network, see **Accessing a Real-Time Service Using a Resource Pool Network**.

Table 1-31 Parameters for private network connection request

Parameter	Description
Service ID	ID of the inference service that needs to be accessed through the private network. Enter the inference service ID obtained in **Prerequisites**.
Network Type	Scenario for the private network connection request. • **User network**: Connect the private network between your VPC and inference service. • **Resource pool network**: Connect the private network between your resource pool and inference service.
VPC	When you select **User network**, choose a VPC for private network connection. If you have not created any VPC and subnet, go to the VPC console to create them. For details, see **Creating a VPC with a Subnet**.
Subnet	When you select **User network**, choose a subnet for private network connection.
Resource Pool	When you select **Resource pool network**, choose a resource pool for private network connection. If you have not created any dedicated resource pool, go to the ModelArts console to create one. For details, see **Creating a Dedicated Resource Pool**.

Parameter	Description
Custom Access Address	You can add custom domain names for service calls within your network. Make sure the address is correct to avoid access issues. Click **Add**, select **Protocol**, and enter **Domain Name (Host)** and **Path**. You can select the default or custom path. Each private network connection instance supports up to 10 custom URLs.

Accessing a Real-Time Service Using Your VPC

1. Before accessing a real-time service using your VPC, create a private network connection for the service and choose **User network** as the scenario. For details, see **Creating a Private Network Connection**. After the request status is **Connection succeeded**, the request is approved.
2. Go to the **Buy ECS** page to purchase an ECS. Select the VPC in **Creating a Private Network Connection**. For details about more parameters, see **Purchasing an ECS**.
3. Log in to the ECS and call the access address and API of the inference service displayed in the private network connection list on the ECS. For details about how to log in to an ECS, see **Logging In to an ECS**.

Accessing a Real-Time Service Using a Resource Pool Network

1. Before accessing a real-time service using your VPC, create a private network connection for the service and choose **Resource pool network** as the scenario. For details, see **Creating a Private Network Connection**. After the request status is **Connection succeeded**, the request is approved.
2. In the **Resource Management** > **Resource Pools** list, click the name of the target resource pool. On the resource pool details page that is displayed, obtain the ID of the resource pool.

   Old navigation path: In the **Resource Management** > **Dedicated Resource Pool** list, click the name of the target resource pool. On the resource pool details page that is displayed, obtain the ID of the resource pool.
3. Go to the **CCE console**. Locate the cluster corresponding to the resource pool based on the resource pool ID, and click the cluster name to access the cluster.
4. Choose **Workloads** on the left, select the corresponding namespace, and view the workload list.
5. Click the workload name to access the desired workload and remotely log in to the container.
6. After accessing the container, run the curl command to access the API. The address is the access address displayed after the private network connection is successfully created.

Figure 1-56 Accessing a real-time service

FAQs

What should I do if the address displayed in the private network connection list is unreachable after I purchase an ECS?

Check whether the ECS security group is open.

Managing Private Network Connections

- Deleting a private network connection request

  If a private network connection is not required anymore, delete it.

  a. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.

  b. Click **Private Network Connections**.

  c. Switch to the **My Requests** tab and click **Delete** in the **Operation** column.

  d. In the displayed dialog box, confirm the information, enter **DELETE**, and click **OK**.

- Approving or rejecting a pending private network connection request

  If you are the private network connection request approver, you can approve or reject requests submitted by third parties in **My Approvals**.

  a. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.

  b. Click **Private Network Connections**.

  c. In the **My Approvals** tab, click **Approve** in the **Operation** column to approve the private network connection request. Click **Reject** in the **Operation** column to reject the request.

- Canceling authorization

  If you are the private network connection request approver, you can cancel the authorization to reject the request again.

  a. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.

  b. Click **Private Network Connections**.

  c. In the **My Approvals** tab, click **Revoke Authorization** in the **Operation** column. In the displayed dialog box, click **OK**.

1.6 Accessing a Real-Time Service Using Different Protocols

1.6.1 Accessing a Real-Time Service Using WebSocket

Context

Traditional HTTP protocols often create and drop connections repeatedly during real-time chats or online gaming, causing delays and using excessive bandwidth. WebSocket is a communication protocol that enables easy two-way data transfer between clients and servers, allowing servers to send data directly to clients. In the WebSocket API, if the initial handshake between the client and the server is successful, a persistent connection will be established between them and data can be transferred bidirectionally. WebSocket is ideal for applications requiring real-time, bidirectional communication, such as online games and real-time chat.

ModelArts uses WebSocket for real-time service deployment. WebSocket only supports real-time services and, when used on ModelArts, the protocol is converted to WebSocket Secure (WSS), which supports one-way authentication.

Key features of WebSocket include:

- Persistent connection: Unlike HTTP, WebSocket maintains a persistent connection between the client and server, reducing the overhead of frequent connection establishment.
- Bidirectional communication: Data can flow in both directions, allowing the server to proactively send data to the client, not just respond to client requests.
- Reduced latency and bandwidth: Real-time data transmission is possible because the connection remains open, minimizing latency and reducing unnecessary HTTP requests and bandwidth usage.

Prerequisites

- The service protocol in the network configuration is set to **WSS** or **WS** during **service information configuration**.
- The image for importing the model is WebSocket-compliant.

Constraints

- WebSocket only supports the deployment of real-time services.
- When a server response exceeds 64 KB, the system splits it into smaller frames as per the protocol rules. Each frame must stay under 64 KB.
- 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 request body must be within the allowed size set by the service's configuration; otherwise, the request will be blocked.
  - Each request must be completed within its configured timeout period, which is recalculated during data transmission.

Obtaining the Authentication Information, Local Path to the Prediction File, and URL of the Real-Time Service

- **Authentication Information**

WebSocket itself does not require additional authentication. WSS supports only one-way authentication, from the client to the server.

Obtain authentication information based on the authentication mode selected during service deployment. Authentication information is not required if no authentication is used.

- **Token authentication**
- **API key authentication**
- **No authentication**

This section uses API key authentication as an example. Before calling a service, **create an API key** and **bind the API key to the real-time service to be accessed**. Open the CSV file automatically downloaded after the API key is created. The **api_key** field in the file is the API key.

- **Local Path to the Prediction File**

  The local path to the prediction 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**).

- **URL of the Real-Time Service**

  The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Basic Information** tab.

  API URL is the 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}***/v1/chat/completions**.

Accessing a Real-Time Service Through Authentication

WebSocket itself does not require additional authentication. WSS only supports one-way authentication, from the client to the server.

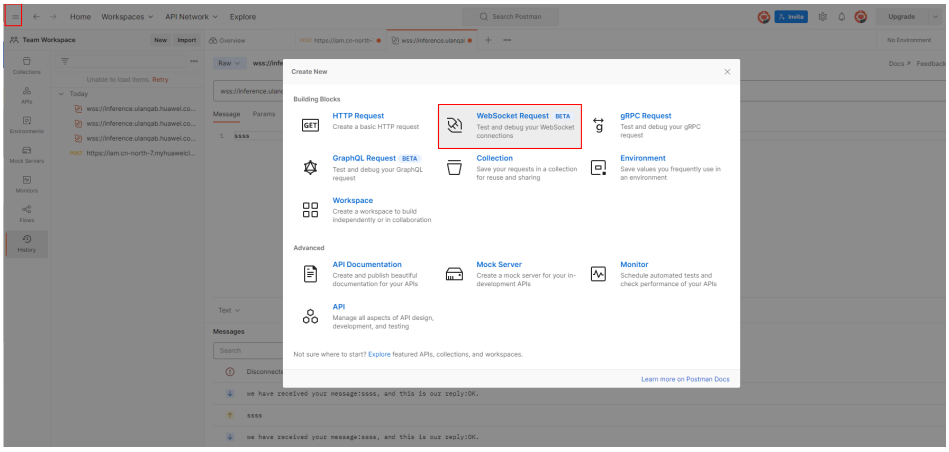
Call the real-time service based on the authentication mode selected during service deployment.

- **Accessing a Real-Time Service Through IAM Token-based Authentication**
- **Accessing a Real-Time Service with No Authentication**
- **Accessing a Real-Time Service Through API Key Authentication**

Creating 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 1-57 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 URL** obtained from **Call Info** in the **Service** tab on the service details page.

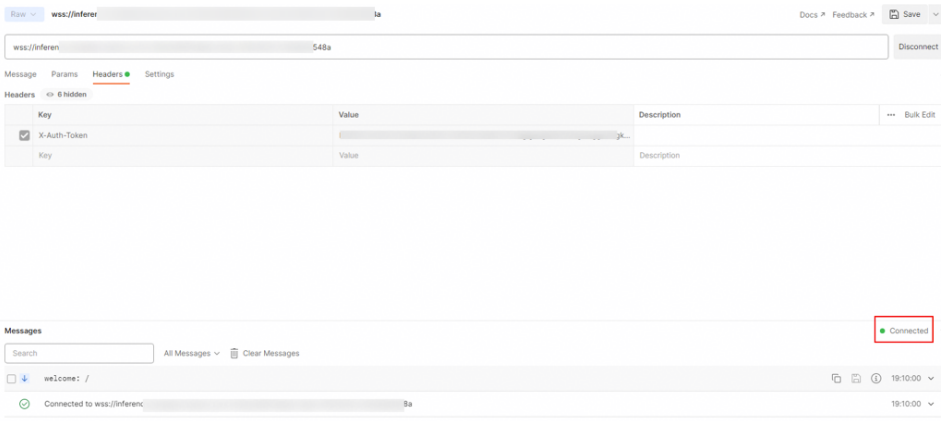
   If there is a finer-grained URL in the custom image, add the URL to the end of the address. If **queryString** is available, add this parameter in the **params** column.

   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. Take API key authentication as an example: In the **Headers** tab, set **KEY** to **Authorization** and **VALUE** to **Bearer** *API key content*.

3. Click **Connect** in the upper right corner to establish a WebSocket connection.
   - 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 WSS.

   The following shows an established WebSocket connection.

Figure 1-58 Connection established



NOTICE

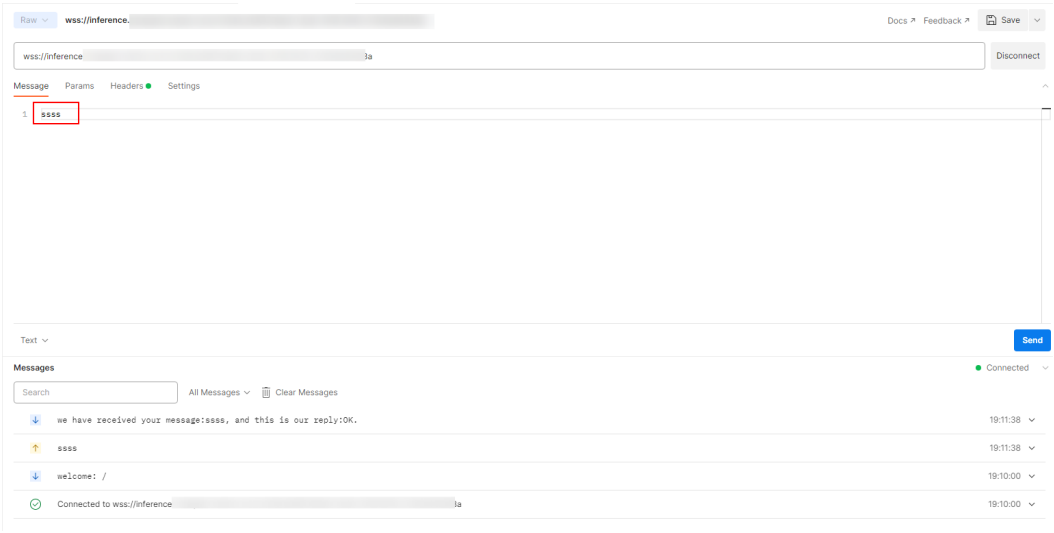
Prioritize verifying the WebSocket service provided by the custom image. Different tools implement varying WebSocket services, which might result in an inability to maintain the connection once established, or the connection could be interrupted after a single request requiring a reconnection. ModelArts solely guarantees that the WebSocket configuration of the custom image remains consistent both before and after deployment on ModelArts (with the exception of differing addresses and authentication methods).

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.

Postman accepts various formats like Text, JSON, XML, HTML, or Binary for sending data. For example, type your text into the text box and press **Send** on the right side to submit the request. The response will appear in the **Response** area below.

Figure 1-59 Sending data



1.6.2 Accessing a Real-Time Service Using 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 unidirectionally push real-time data to a client, for example, a real-time news update or stock prices.

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.

Key features of SSE include:

- Easy to use: SSE is based on the HTTP protocol and is straightforward to implement. No complex configurations or additional libraries are needed, and data can be pushed in real-time over standard HTTP connections.
- Automatic reconnection: SSE supports automatic reconnection. If the connection is interrupted, the client automatically attempts to reconnect, ensuring continuous data delivery.
- Unidirectional communication: SSE is unidirectional, meaning the server can send events to the client, but the client cannot send data back to the server through the same connection.
- Low resource usage: SSE uses HTTP connections, making it less resource-intensive compared to other real-time communication protocols like WebSocket. This makes SSE ideal for lightweight real-time data push scenarios.

Prerequisites

- The image for importing the model in **Configuring Deployment Settings** is SSE-compliant.

- The service protocol in the network configuration is set to **HTTP** or **HTTPS** during **service information configuration**.

Constraints

- SSE supports only the deployment of real-time services.
- It supports only real-time services deployed using models imported from custom images.
- 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 request body must be within the allowed size set by the service's configuration; otherwise, the request will be blocked.
  - Each request must be completed within its configured timeout period.

Obtaining the Authentication Information, Local Path to the Prediction File, and URL of the Real-Time Service

- **Authentication Information**

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

  Obtain authentication information based on the authentication mode selected during service deployment. Authentication information is not required if no authentication is used.

  - **Token authentication**
  - **API key authentication**
  - **No authentication**

  This section uses token authentication as an example. 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.

- **URL of the Real-Time Service**

  The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Service** tab.

  API URL is the 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}***/v1/chat/completions**.

Calling a Real-Time Service via SSE

1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 8.11.1 is recommended.
2. Open Postman and set parameters on the Postman interface.
   - Select a POST task and copy the obtained API URL to the POST text box. Add **/v1/sse** to the end of the address to access the service via SSE. 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. Take token authentication as an example: On the **Headers** tab page, set **Key** to **X-Auth-Token** and **Value** to the user token.

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

Figure 1-60 Parameter settings

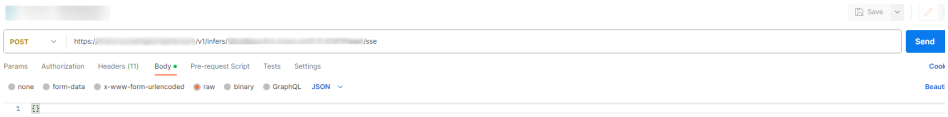


Figure 1-61 Response header **Content-Type**

– Select **raw** and **JSON (application/json)**, and enter the request body in the text box below. The format and contents of the request body depend on the model being used. The platform does not process the input data in any way.

  Example request body:

```
{
   "model": "test",
   "messages": [
      {
         "role": "user",
         "content": " Who are you?"
      }
   ],
   "max_tokens": 100,
   "top_k": -1,
   "top_p": 1,
   "temperature": 0,
   "ignore_eos": false,
   "stream": false
}
```


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

   Figure 1-34 shows an example of prediction result. The returned contents and format depend on the model being used.

1.6.3 Accessing a Real-Time Service Using HTTP or HTTPS

Context

HTTP is based on the request-response mode. That is, the client initiates a request, and the server returns a response. The server cannot proactively push data. HTTPS enhances security based on HTTP and is widely used in scenarios that require high data privacy and integrity.

HTTP sends content in plaintext and does not provide data encryption in any mode. If an attacker intercepts the packets transmitted between the web browser and website server, the attacker can directly read the information in the packets. Therefore, HTTP is not suitable for transmitting sensitive information. HTTPS communicates through HTTP, but uses SSL/TLS to encrypt data packets. HTTPS is used to authenticate website servers, protect privacy, and ensure integrity of exchanged data.

HTTP/HTTPS features:

- Statelessness: By default, session states are not retained. Each request is processed independently. Stateful session management (such as user login status) can be implemented through cookies, sessions, or tokens.
- Flexibility and scalability: transmission of multiple data formats, such as text (JSON/XML) and binary (images and files). You can also use the header field for scalability, such as cache control, content negotiation, and cross-origin resource sharing (CORS).
- Wide compatibility: Native support for all typical browsers, programming languages, and frameworks, without the need to upgrade specific libraries or protocols. HTTP and HTTPS are applicable to common scenarios such as web pages, RESTful APIs, and file transfer.

Prerequisites

- The service protocol in the network configuration is set to **HTTP** or **HTTPS** during **service information configuration**.
- The image selected for the real-time service must support HTTP/HTTPS.

Constraints

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 request body must be within the allowed size set by the service's configuration; otherwise, the request will be blocked.
- Each request must be completed within its configured timeout period, which is recalculated during data transmission.

Obtaining the Authentication Information, Local Path to the Prediction File, and URL of the Real-Time Service

- **Authentication Information**

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

  Obtain authentication information based on the authentication mode selected during service deployment. Authentication information is not required if no authentication is used.

  - **Token authentication**
  - **API key authentication**
  - **No authentication**

  This section uses token authentication as an example. 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.

- **Local Path to the Prediction File**

  The local path to the prediction 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**).

- **URL of the Real-Time Service**

  The API URL and input parameters of the real-time service: To obtain them, choose **Model Inference** > **Real-Time Inference** on the console, click the target real-time service, and obtain the information from **Call Info** in the **Service** tab.

  API URL is the 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}***/v1/chat/completions**.

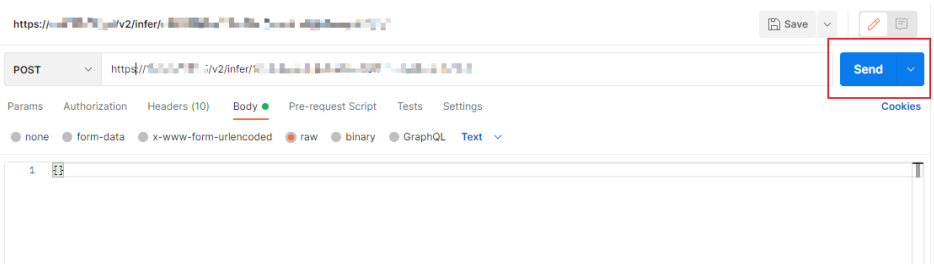
Calling a Real-Time Service via HTTP/HTTPS

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

1. Download Postman and install it, or install the Postman Chrome extension. Alternatively, use other software that can send POST requests. Postman 8.11.1 is recommended.
2. Open Postman, select a POST task, and copy the API URL obtained in **Obtaining the Authentication Information, Local Path to the Prediction File, and URL of the Real-Time Service** to the POST text box.

   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. Take token authentication as an example: On the **Headers** tab page, set **Key** to **X-Auth-Token** and **Value** to the user token.

Figure 1-62 Using Postman for prediction



3. Click **send** to send the request. The result will be displayed in **Response**.

1.7 Managing Real-Time Services

1.7.1 Viewing Details About a Real-Time Service

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

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.

   Click **Quick Start** in the upper right corner to view the online inference process guide.

   In the search box on the real-time service list page, you can search for services by service name, service ID, service status, service creator, service authentication mode, and ID of the resource pool where the service is deployed.

2. Click the target service name to access its details page. Switch between tabs on the details page to view more details. For details, see **Table 1-32**.

Table 1-32 Service details

Parameter	Description
Service	Displays the topology, basic information, network configuration, traffic weight, and advanced configuration of the service. For details about the service topology, see **Viewing the Service Deployment Topology**. Click to copy the API URL for calling the service.

Parameter	Description
Deployment	Displays the service deployment information. You can switch between cards on the left to view the information of different deployments. In the deployment information area, click the resource pool name to access its details page and view the configuration. For details, see **Viewing Details About a Dedicated Resource Pool**. Click **View Details** next to the number of deployment replicas to view the service instance details. • You can delete instances as needed or scale deployed instances. For details, see **Scaling a Real-Time Service Deployment**. • In the deployment replica list, you can click the number of pods to view pod details. You can view pod events and logs in the pod operation column and delete pods as required. If the resource pool is a public resource pool, you cannot delete and recreate instances or delete pods. Click **Modify Configuration** next to **Traffic Weight** to modify the traffic weight of the service deployment. For details, see **Viewing/Modifying Traffic Weights**. Click the version count to see all deployed versions. You can switch between or delete these versions. The current version cannot be deleted. In the deployment card on the left, you can perform operations such as upgrade, stop, and scale. For details, see **Managing the Lifecycle of a Real-Time Service Deployment**.
Prediction	Performs real-time prediction on this page. For details, see **Using the Prediction Feature**.
Monitoring	Shows monitoring data for the service. For details, see **Viewing Performance Metrics of a Real-Time Service on ModelArts**.
Cloud Shell	You can use Cloud Shell provided by the ModelArts console to log in to the instance container of a running real-time service. For details, see **Using Cloud Shell to Debug a Real-Time Service Instance Container**.

Parameter	Description
Events	Service events: Records key activities like starting, stopping, updating, or recovering services. Data is retained for one month and automatically cleared thereafter. Pod events: Records lifecycle events and exceptions for pods within the Kubernetes cluster. Data is retained for one hour and automatically cleared thereafter. For details about how to view events of a service, see **Viewing Events of a Real-Time Service**.
Logs	Runtime logs are exported to Log Tank Service (LTS). LTS automatically creates log groups and streams and caches logs for seven days by default. You can search for and analyze runtime logs. • Log search: Search logs using specific keywords or phrases. Narrow your results by selecting a specific time range to find events and issues during that period. For details, see **Searching for Logs**. • Statistical charts: After sending logs to LTS, use **Using SQL Analysis Syntax** to find important log data and view the results as statistical charts. For details, see **Visualizing Logs in Statistical Charts**. • Log analysis: Before searching for analyzing logs, set up structured data and indexing for them. For details, see **Setting Cloud Structuring Parsing**. • Real-time logs: Once you connect your real-time service logs to LTS, they will be sent every minute. You can view these updates from the **Real-Time Logs** tab, where you can also easily search and analyze the data. For details, see **Viewing Real-Time Logs**.

Parameter	Description
Intelligent O&M	As foundation model deployment scales and the complexities of cross-node deployment and load balancing grow, traditional resource scaling strategies based on native Kubernetes Horizontal Pod Autoscaler (HPA) can no longer satisfy the need for refined adjustments to the prefill (P) and decode (D) instance ratio. In real-world operations, users often face low resource utilization or performance bottlenecks due to sub-optimal P/D ratios, yet existing tools lack the ability to provide dynamic ratio recommendations based on simulation algorithms. To resolve this, ModelArts resource pools now support the installation of the HRA plugin. By utilizing simulation algorithms to calculate and display the optimal P/D ratio recommendation, users can manually adjust instance ratios based on real-time metric analysis, without relying on auto scaling, to achieve highly efficient inference service deployment within their resource capacity. Enable **Monitoring** and click the edit button to view the optimal inference unit ratio calculated based on the intelligent algorithm policy. When a real-time service uses a physical resource pool with the HRA plugin installed, and the model assets of the real-time service contain the "dynamic ratio recommendation" label, inference unit ratio detection is supported.
Private Network Connections	Displays the private network connection requests that need to be approved by the current account. ModelArts offers private network connection. When you create a private network connection request, it automatically sets up a VPCEP to connect your VPC with the real-time inference service securely. For details, see **Accessing a Real-Time Service Through a Private Network**.
Tags	Displays tags that have been added to the service. You can add, modify, and delete tags. For details about how to use tags, see **Using TMS Tags to Manage Resources by Group**.

Viewing the Service Deployment Topology

- Deployment topology

  During service deployment, you can view the topology corresponding to the current deployment configuration.

Figure 1-63 Deployment topology

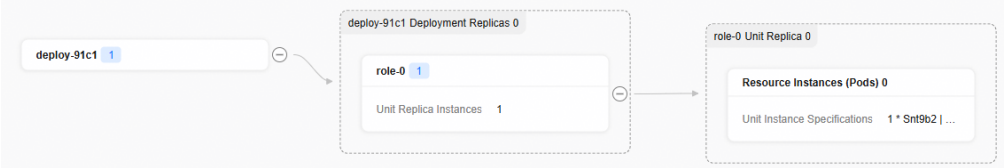


Table 1-33 Deployment topology description

Topology Layer	Deployment Information	Description
1	deploy	Deployment name of the current service.
2	Deployment replica	Deployment replicas configured for the current service deployment and the number of unit replica instances
3	Unit replica	Unit replicas configured for the corresponding deployment replica and the number of resource instances

- Service topology

  After deploying a real-time service, you can view the service topology on the service details page.

Figure 1-64 Service topology

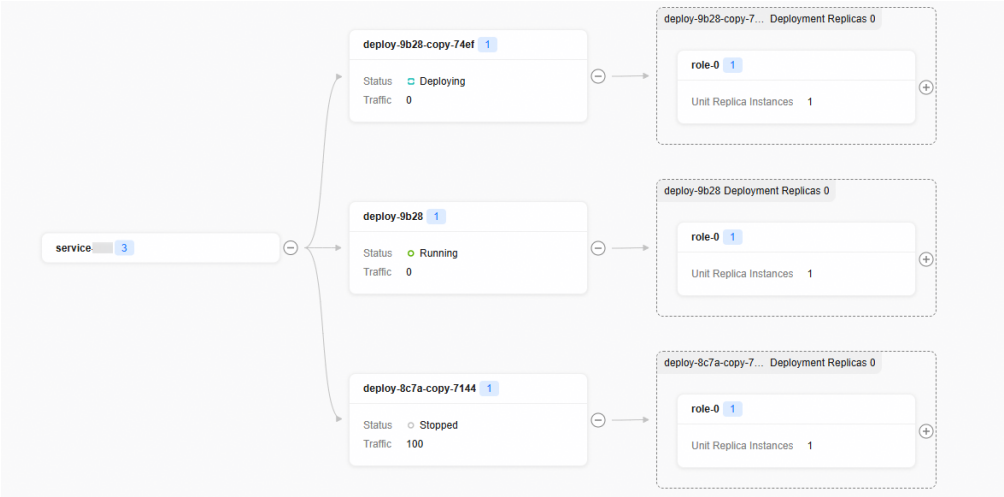


Table 1-34 Service topology description

Topology Layer	Deployment Information	Description
1	service	Name of the current service
2	deploy	Deployment of the current service, as well as the status and traffic of each deployment

Topology Layer	Deployment Information	Description
3	Deployment replica	Deployment replicas of the corresponding service deployment and the number of unit replica instances
4	Unit replica	Unit replicas of the corresponding deployment replica and resource instance information

1.7.2 Managing the Lifecycle of a Real-Time Service Deployment

For services deployed through the public resource pool, you cannot manage their lifecycle independently. This includes actions like adding, starting, stopping, changing versions, pausing, deleting, upgrading, cloning, or scaling real-time service deployments.

Adding Deployment Configurations

For a deployed real-time service, add deployment configurations to match service changes and upgrade the service.

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Choose **More** > **Add Deployment** in the **Operation** column of the target service.
3. Configure deployment information by referring to the parameter description in **Configuring Deployment Settings**.
4. Click **Confirm Deployment** and add a deployment task as prompted.

Starting a Service Deployment

You can start services in the **Stopped** or **Failed** state. Services in the **Deploying** state cannot be started. Start the service deployment by following these steps. Once started, the deployment will be in the **Running** state.

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Locate the target deployment card and click **Start**. In the displayed dialog box, click **OK** to start the deployment.

Stopping a Service Deployment

Deployments in the **Running** or **Abnormal** states can be stopped. You can stop them using one of these methods:

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Locate the target deployment card and click **Stop**. In the displayed dialog box, click **OK** to stop the deployment.

Switching the Real-Time Service Version

Change the real-time service version based on service changes.

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Locate the target deployment card and choose **...** > **Change Version** to switch the current version.
4. In the displayed dialog box, select the target version of the real-time service, click **Switch to This Version** in the **Operation** column, confirm the version comparison information, and click **OK**.

   You can also delete the deployed versions in the dialog box. The current version cannot be deleted.

Interrupting a Service Deployment

Interrupt a service deployment in the **Deploying** state to quickly stop it. You can interrupt a service deployment if it is severely faulty and needs to be recovered immediately, resources need to be quickly released to deploy a service with a higher priority, or fast iteration is required in the test environment.

- If a deployment in the **Deploying** state is interrupted, the deployment status changes to **Stopped**, related resources will be released, and the interruption operation will be recorded.
- If a deployment in the **Upgrading** state is interrupted, the deployment status changes to **Running**.
- If an exception occurs, such as unauthorized permission, invalid service ID, or version number does not exist, the API returns an error message.

To interrupt a service, follow these steps:

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Locate the target deployment card and click **Interrupt**. In the dialog box that is displayed, click **OK** to stop the deployment.

Deleting a Service Deployment

If a deployment configuration is no longer used, you can delete it to release resources.

WARNING

The deletion cannot be undone.

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Locate the deployment card to be deleted and choose **...** > **Delete**.
4. In the displayed dialog box, enter **DELETE** and click **OK**.

Deleting a Real-Time Service

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

- A service cannot be deleted without agency authorization.
- 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.

WARNING

The deletion cannot be undone.

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. Delete the real-time service in any of the following ways:

- Choose **More** > **Delete** in the **Operation** column of the target real-time service. In the displayed dialog box, enter **DELETE** and click **OK** to delete the service.
- Select services in the real-time service list and click **Delete** in the upper left corner of the list. In the displayed dialog box, enter **DELETE** and click **OK** to batch delete services.
- Click the target service name. On the displayed service details page, click **Delete** in the upper right corner. In the displayed dialog box, enter **DELETE** and click **OK** to delete the service.

Other Deployment Operations

- **Upgrading a Real-Time Service Deployment**: To update a real-time service deployment, modify its configuration settings to include a new deployment version.
- **Scaling a Real-Time Service Deployment**: Once a real-time service is running, its resource needs might shift as services evolve. ModelArts allows you to deploy and scale these services dynamically. You can modify instance counts or resource settings to match current demand.
- **Cloning a Real-Time Service Deployment**: This feature allows you to duplicate an existing real-time service's deployment setup, adjust it as needed, and quickly create a new deployment.

1.7.3 Upgrading a Real-Time Service Deployment

To update a real-time service deployment, modify its configuration settings to include a new deployment version.

Changing resource settings, environment settings, deployment management settings, or advanced settings (key settings and custom metric collection) will cause a rolling upgrade. This may cause redeployment.

WARNING

Improper upgrade operations will interrupt deployments during the upgrade.

Prerequisites

You can upgrade a service deployment in the **Running**, **Abnormal**, **Alarm**, or **Stopped** state.

Upgrade Procedure

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Locate the card of the service you want to upgrade and click **Upgrade**.
4. On the **Upgrade Deployment** page, configure parameters. For details, see **Configuring Deployment Settings**.
5. Confirm the information and click **Confirm Deployment** to submit the task. "Upgrade deployment task submitted" is displayed in the upper right corner. The task is submitted, and the real-time service list page is displayed.

   When the maximum number of upgrades is reached, a dialog box appears requesting version deletion. Click **Delete** in the **Operation** column and follow the prompts to delete them. The current version cannot be deleted.

1.7.4 Scaling a Real-Time Service Deployment

Once a real-time service is running, its resource needs might shift as services evolve. ModelArts allows you to deploy and scale these services dynamically. You can modify instance counts or resource settings to match current demand.

ModelArts provides manual scaling and auto scaling to meet different user requirements. Only the number of instances of a single deployment can be changed.

- **Manual scaling** allows you to manually change the number of instances of a single deployment.
- **Auto scaling** allows you to configure scaling rules to adjust instance count. This helps you use your resources more efficiently. Auto scaling allows you to configure scaling policies to add instances when the traffic is high, and reduce them when the traffic is low. This helps you use your resources more

efficiently. Currently, only deployments on dedicated resource pool support auto scaling.

Table 1-35 Comparison between manual and auto scaling

Feature	Manual Scaling	Auto Scaling
Configuration Method	Manual	Automatic
Operation	Modify the number of instances	Set scaling rules
Execution	Executed after manual configuration	Triggered by schedule or metrics
Impact of Scaling Failure	The number of instances reverts to the previous value.	The number of instances changes to a specific value.

Constraints

- To set scaling rules, the service must be in the **Stopped**, **Running**, or **Alarm** state.
- Real-time services allow up to 10 periodic-based scaling rules. You can add only one scaling rule per metric.

Manual Scaling

Step 1 Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.

Step 2 Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.

Step 3 Locate the target deployment card and choose **...** > **Scale**.

Step 4 Click **Manual Scaling**. In the displayed dialog box, set the number of deployment replicas after scaling. The minimum value is **1**. Click **OK**.

Check the instance list to view their status after manual scaling.

To delete a specified instance, click **Delete** in the **Operation** column in the deployment replica list. Choose how you want to delete it, enter **DELETE**, and click **OK**. Currently, **Delete and re-create** is supported.

WARNING

Deleting a replica reduces the total count permanently. Exercise caution when performing this operation.

----End

Auto Scaling

Step 1 Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.

Step 2 Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.

Step 3 Locate the target deployment card and choose **...** > **Scale**.

Step 4 In the **Auto Scaling Rule** area, click **Configure Scaling Rule**. In the displayed dialog box, set automatic scaling rules.

Real-time services allow up to 10 periodic-based scaling rules. You can add only one scaling rule per metric.

A periodic-based scaling rule cannot be added repeatedly. For details, see **Table 1-36**.

Table 1-36 Scaling rule parameters

Scaling Rule Type	Description
Periodic	Automatically adjusts the number of nodes in the pool within a set timeframe, improving resource efficiency and lowering expenses. • **Trigger Time**: Specify a time as required. This indicates the local time where the nodes are deployed. • **Target Instances**: Target number of nodes in the node pool after auto scaling.

Step 5 Click **OK**.

Once you add a rule, activate it in the **Scaling Rules** section. When enabled, the system adjusts your service deployment size automatically at the set time. Check the scaling history at the bottom of the page.

Click **Delete** in the **Operation** column to delete the created scaling rule.

----End

1.7.5 Cloning a Real-Time Service Deployment

This feature allows you to duplicate an existing real-time service's deployment setup, adjust it as needed, and quickly create a new deployment.

Prerequisites

A service has been deployed.

Constraints

- Cloning a service copies its deployment but does not include its status or history.

- The cloned service needs to be deployed again, which may occupy new resources.
- Real-time services cannot be cloned across regions.

Procedure

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the target service name. On the service details page that is displayed, switch to the **Deploy** tab.
3. Clone the real-time service and deploy it to either the existing one or a different service.
   - Cloning a real-time service deployment to the current service

     Locate the deployment card you want to clone, choose **...** > **Clone to This Service**, clone the deployment to the current service, and click **OK**.

     On the displayed page, modify the deployment configuration and clicking **Confirm Deployment** to submit the cloning task. For details about the parameters, see **Configuring Deployment Settings**.
   - Cloning a real-time service deployment to a different service
     i. Locate the deployment card you want to clone, choose **...** > **Clone to Other Services**, and clone the deployment to another service.
     ii. In the displayed dialog box, select the target service for cloning and click **OK**.
     iii. On the displayed page, modify the deployment configuration and clicking **Confirm Deployment** to submit the cloning task. For details about the parameters, see **Configuring Deployment Settings**.

1.7.6 Modifying a Real-Time Service

A deployed service can be upgraded to match service changes.

You can modify the basic information about a service on the service management 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 services during the upgrade.

Modifying Service Information on the Service Management Page

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Choose **More** > **Modify** in the Operation column of the target service, modify the basic information about the service based on **Procedure**, and click **Confirm Modification** to submit the modification task.

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.

1.7.7 Modifying the Name of a Real-Time Service

You can change the name of a created real-time service to make management easier and upgrade the service.

Prerequisites

The service is not in the **Deploying**, **Stopping**, **Upgrading**, or **Deleting** status.

Constraints

After modifying the service name, the changes will only apply to new pods created through operations such as restarts or new deployments. Existing pods will remain unchanged.

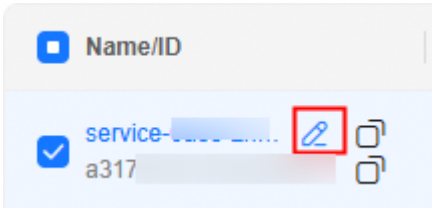
For example, if you modify the service name and then restart the service, you can view the newly reported container metrics by **service_name** or **task_name** in AOM. For a new deployment, the values of **service_name** and **task_name** in the container metrics of the new pod will also be the new service name.

The service name can contain 1 to 128 characters, including letters, digits, hyphens (-), and underscores (_).

Procedure

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. In the service list, click to the right of the real-time service name, edit the name, and click **OK**.

Figure 1-65 Modifying a service name



1.8 Advanced Functions

1.8.1 Asynchronous Inference

When handling AI-generated content or demanding tasks like HD image processing and long video analysis, the standard request-response approach struggles. Clients face long waits that can lead to network timeouts, servers get overwhelmed by sudden traffic spikes, and valuable GPU resources sit idle during

delays. Decoupling task submission from result retrieval ensures high system availability and efficient resource usage.

ModelArts enables building asynchronous inference services using tasks. It offers standard APIs for submitting tasks asynchronously, checking their status, and retrieving results. This approach separates long-running tasks from synchronous processes. A task queue manages dynamic resource scheduling and load balancing, boosting system performance, enhancing reliability, improving user experience, and lowering computing costs per task.

This section describes how to deploy a model as an asynchronous real-time service on ModelArts and call the service. For large video files, deploy and call an asynchronous real-time service by following this section. For images and rather small video files, follow steps in **Deploying a Real-Time Inference Service Using a Single Node**.

Prerequisites

- You have prepared data as instructed in **Preparations**.
- Your account is not in arrears to ensure available resources for running services.
- You have configured the inference service information as instructed in **Configuring Service Information**.

Notes

Resource pools allocate quota for real-time services even when they are **Abnormal** or **Stopped**. If the quota is insufficient and no more services can be deployed, delete some abnormal services to release resources.

- Quota calculation:

  The quota stays the same when you deploy real-time services with a dedicated resource pool. It only changes if you create, modify, or delete a resource pool.

- Usage metering:

  Deploying real-time services in a dedicated resource pool is not metered. Only the usage of the dedicated resource pool itself is metered.

- Mounting SFS Turbo:

  Before mounting an SFS Turbo file system to a real-time service, **associate the dedicated resource pool network with the file system**.

Procedure

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.
2. In the real-time inference list, click **Deploy**.
3. On the service deployment page, set the following key parameters. For details about other parameters, see **Procedure**.

Table 1-37 Asynchronous service parameters

Parameter	Sub-Parameter	Description
Basic Information	Service Call Mode	Select **Asynchronous**. The call mode cannot be modified after the service is created.
	Max Tasks per Service	The maximum number of tasks that can be created for a single service. The value ranges from 0 to 10,000. Tasks in **deleted**, **succeeded**, or **timeout** state do not consume task quota. You can obtain the task status by calling the API for **viewing task details**.
Network Settings	Service Protocol	Asynchronous inference services support HTTPS and HTTP. Currently, asynchronous services do not support traffic mirroring.
	More Settings	• **Public Network Access**: Specifies whether to allow external network access to the real-time service. If this function is enabled, external networks can access the service. View the API URL on the service details page. If this function is disabled, real-time services cannot be accessed over the Internet. • **Auto-approved Private Network Connection**: Specifies whether private network connection to real-time services requires approval. When this function is enabled, private network connection requests from third-party users will be approved automatically. If this function is disabled, approval is required. For details about how to access a service through a private network, see **Accessing a Real-Time Service Through a Private Network**. • **Access Control**: Once selected, you can specify a whitelist or blacklist for access control. Whitelist: Only user IP addresses from the CIDR blocks configured here are allowed access. You can add up to ten regular expressions. Blacklist: Only user IP addresses from the CIDR blocks configured here not allowed access. You can add up to ten regular expressions.
High Availability Settings	More settings	• **Request Size Limit**: maximum size of a request for a single service. Value range: **1** to **50**. Unit: MB. • **Request Timeout (s)**: timeout interval for a service prediction request. The value ranges from **1** to **1200**. Unit: second.

4. Click **Next** to configure service deployment. The following table describes the key parameters. For details about other parameters, see **Configuring Deployment Settings**.

   Table 1-38 Asynchronous service deployment parameters

Parameter	Sub-Parameter	Description
Resource Settings	Max Concurrent Tasks per Replica	This parameter is mandatory when the **Service Call Mode** is set to **Asynchronous**. Maximum number of tasks that can be concurrently processed by a single replica. The value ranges from 1 to 100. As long as a task is not deleted from the replica container, the quota is occupied.

5. Click **Next**. On the **Deploy Real-Time Service** > **Confirmation** page, confirm the configuration information and click **Confirm Deployment**.

   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.

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

   NOTE

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

Accessing an Asynchronous Real-Time Service

On the **ModelArts console**, choose **Model Inference** > **Real-Time Inference** in the left navigation pane. On the displayed real-time inference list, click **Call Service** in the **Operation** column of the target service to view the call information.

Table 1-39 Call information for an asynchronous real-time service

Type	URL Format	URL Example
POST (creating a task)	https://{public-network-address}/v2/async-infer/{service-ID}	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks
DELETE (deleting a task)	https://{public-network-address}/v2/async-infer/{service-ID}/{task_id} **task_id**: returned after a task is created.	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks/{task_id}

Type	URL Format	URL Example
GET (obtaining tasks)	https://{public-network-address}/v2/async-infer/{public-network-address}/tasks	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks
PUT (updating a task)	https://{public-network-address}/v2/async-infer/{public-network-address}/tasks/{task_id}	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks/{task_id}
GET (querying task details)	https://{public-network-address}/v2/async-infer/{public-network-address}/tasks/{task_id}	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks/{task_id}
POST (starting a task)	https://{public-network-address}/v2/async-infer/{public-network-address}/tasks/{task_id}/start **task_id**: returned after a task is created.	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks/{task_id}/start
POST (stopping a task)	https://{public-network-address}/v2/async-infer/{public-network-address}/tasks/{task_id}/stop **task_id**: returned after a task is created.	https://100.XX.XXX.XXX/v2/async-infer/testxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/tasks/{task_id}/stop

Calling an Asynchronous Real-Time Service

After an asynchronous service is deployed, verify its performance using the **Call** tab on the service details page. The service call tab offers the rest client. You can choose your service's request method and input the prediction path.

In the **Params** column, you can set path parameters such as task_id, or set the limit, offset, and status parameters for querying the task list.

In the **Body** tab, set data format to **raw** to send original text data. Ensure that the data complies with the API specifications and the JSON format is correct.

You can enter header information in **Headers**, for example, API key authentication information. Replace **{API Key}** with your own API key. Click **Call** to send a request.

Deleting the authorization key pair makes the system switch to IAM token authentication automatically.

For details about how to set the body and request header on the inference page, see **How Do I Fill in the Request Header and Request Body When a ModelArts Real-Time Service Is Running?**

NOTE

The status of an asynchronous service affects the calling of task APIs.

Using Cloud Shell to Debug a Real-Time Service Instance Container

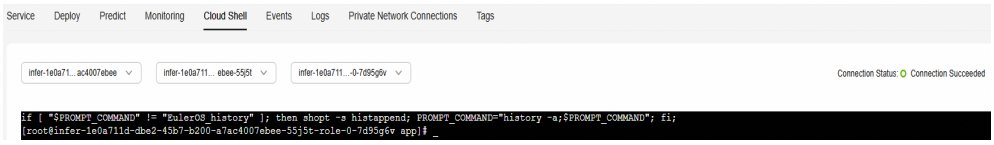
You can use Cloud Shell provided by the ModelArts console to log in to the instance container of a running real-time service.

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

1. Log in to the ModelArts console. In the navigation pane on the left, choose **Model Inference** > **Real-Time Inference**.
2. Click the real-time service name or ID to access its details page.
3. In the **Cloud Shell** tab, select the deployment, instance, and Pod. If the connection status changes to **Connection Succeeded**, you have logged in to the instance container.

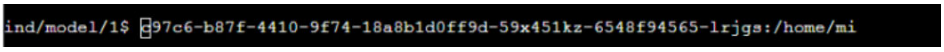
   If the server disconnects due to an error or remains idle for 10 minutes, you can select **Reconnect** to regain access to the Pod.

   Figure 1-66 Cloud Shell

   If you encounter a path display issue when logging in to Cloud Shell, press **Enter** to resolve the problem.

   Figure 1-67 Path display issue

4. After logging in to the container, execute the necessary debugging commands in its terminal. Example:

   NOTE

   The following is for reference only. The actual log paths and service health check methods depend on your service configuration. Refer to your image settings and container startup commands for details.

   View logs:

   ```
   tail -f /var/log/app.log
   ```

   Check the service status:

   ```
   systemctl status app
   ```

   Run a custom script:

   ```
   ./debug_script.sh
   ```


5. After the debugging, exit the container:
   ```
   exit
   ```
   After returning to the Cloud Shell terminal, you can view the debugging result or log file.

Follow-Up Operations

- For details about how to view real-time service details, see **Viewing Details About a Real-Time Service**.
- For details about how to modify, stop, and delete a real-time service, see **Managing the Lifecycle of a Real-Time Service Deployment**.

1.8.2 Scheduling Policy

Overview

In Huawei Cloud resource management scenarios, users configuring inference services may need to optimize scheduling policies to enhance service performance or resource utilization. For instance, to ensure high availability, users typically select a high-availability scheduling policy, which requires service replicas (pods) to be distributed as evenly as possible across different nodes to minimize the impact of individual failures. However, in practice, users may find that existing scheduling configuration options lack the flexibility required for specific business needs, such as scenarios needing finer control over scheduling priorities, or those requiring a compact scheduling policy to maximize resource utilization. Faced with this challenge, users often wonder: How can scheduling policies be configured more flexibly on Huawei Cloud to adapt to diverse service requirements?

To address this, ModelArts offers three distinct scheduling policies for real-time inference services in dedicated resource pools: high-availability scheduling, compact scheduling, and affinity scheduling. By flexibly adjusting the distribution logic of service instances across cluster nodes, these policies satisfy three core service demands: highly reliable operations, efficient resource utilization, and custom deployment control.

Supported Scheduling Policies

Inference services support the coordination of three scheduling policies: high-availability scheduling, compact scheduling, and affinity scheduling.

- High-availability scheduling: Requires service replicas (pods) to be distributed as evenly as possible across different nodes to minimize the impact of individual failures.
- Compact scheduling: Also known as the binpack scheduling policy. It reduces resource fragmentation to maximize utilization and takes effect across the entire cluster.
- Affinity scheduling: node affinity and anti-affinity.

Table 1-40 Comparison of scheduling policies

Dimension	High-Availability Scheduling	Compact Scheduling	Affinity Scheduling
Core purpose	Resists individual failures; guarantees stable and continuous operations.	Consolidates idle resources; improves overall resource utilization.	Customizes deployment scope; achieves dedicated service control and isolation.
Use case	Online core production business; real-time, high-concurrency inference.	Testing and debugging; gray services; low-traffic, non-core services.	Model warmup deployment; hardware binding; service security isolation.
Instance distribution	Dispersed and distributed across multiple different physical nodes.	Concentrated and gathered into the minimum number of nodes.	Target-deployed or target-avoided based on custom rules.
Compatible resource pools	Dedicated resource pools only.	Dedicated resource pools only.	Dedicated resource pools only.
Failure risk	Dispersed risk; individual failures have a minimal impact.	Concentrated risk; node failure can easily batch affect services.	Depends on the status of selected nodes; strong rules offer low fault tolerance.
Resource utilization	Moderately stable.	Highest and optimal.	Controllable on demand.
Core constraints	Number of replicas ≥ 2; relies on sufficient cluster nodes.	Not for core production; limited to node pools of the same specification.	Strong rules can easily cause deployment failure; limited number of nodes can be selected.
Basic prerequisites	Dedicated resource pool runs normally; replica count meets the standard.	No strict high-availability requirements; uniform node specifications.	Target nodes are healthy and available; warmup/node partitioning completed in advance.
Quick configuration points	Raise scheduling priority; configure rolling upgrades and self-healing.	Turn off affinity scheduling; lower scheduling priority.	Enable scheduling; select type and strength; specify nodes.

High-Availability Scheduling

Concepts

High-availability scheduling is the default recommended policy for real-time inference services. Its core objective is to eliminate individual failure and guarantee continuous service availability. This policy is ideal for core production scenarios with zero-downtime tolerances and stringent stability requirements, such as online intelligent customer service, real-time financial risk control, and autonomous driving decision-making inference. By distributing service replicas across different physical nodes, racks, or even data centers, it prevents a single-node or single-rack failure from causing total service unavailability. Furthermore, it supports seamless transitions during rolling upgrades, ensuring zero service interruption.

Configuration entry

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, select **HA scheduling** in **Resource Settings** > **Scheduling Policy**. For details about how to deploy a real-time service, see **Configuring Deployment Settings**.

Constraints

- You can select either **HA scheduling** or **Compact scheduling**. By default, **HA scheduling** is enabled.
- If both **HA scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence.

Compact Scheduling

Concepts

The core objective of compact scheduling is to maximize resource utilization and reduce deployment costs. Enable bin packing for cluster workloads. The scheduler will prioritize placing pods on nodes with higher resource consumption to reduce idle resource fragmentation and improve cluster resource utilization.

Use cases

This mode works well for situations needing moderate stability and lower costs, like non-core tests, offline inference transitions, and edge deployments with few resources. Common uses are checking model functions, low-traffic tests, and light inference on edge devices.

Configuration entry

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, select **Compact scheduling** in **Resource Settings** > **Scheduling Policy**. For details about how to deploy a real-time service, see **Configuring Deployment Settings**.

Constraints

- You can select either **HA scheduling** or **Compact scheduling**. By default, **HA scheduling** is enabled.
- When both **Compact scheduling** and **Affinity Scheduling** are enabled, **Affinity Scheduling** takes precedence.

Affinity Scheduling

Concepts

Affinity scheduling allows you to control where service replicas run by using node affinity or anti-affinity rules. This helps meet specific deployment needs. You can configure the node affinity type and strength to flexibly schedule workloads in a resource pool. If no nodes are specified, the pods will be randomly scheduled according to the default cluster scheduling policy.

Configuration entry

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, select **Affinity Scheduling** in **Unit Settings** > **More Settings**. In the dialog box displayed on the right, configure the affinity scheduling policy. For details about how to deploy a real-time service, see **Configuring Deployment Settings**.

Table 1-41 Affinity scheduling configuration

Affinity Type	Strength	Use Case
Node affinity	Weak affinity	Prioritizes deployment to specified nodes. If node resources are insufficient, scheduling can fall back to other nodes, balancing cache reuse with resource flexibility.
	Strong affinity	Enforces deployment to specified nodes. Ideal for dedicated nodes with pre-warmed models, nodes with local hardware encryption cards mounted, or edge nodes bound to specific peripherals (such as cameras or sensors), ensuring model cache reuse and exclusive hardware resource utilization.
Node anti-affinity	Weak affinity	Avoids deployment to specified nodes as much as possible, but allows compromised deployment if no other nodes are available, balancing isolation requirements with resource availability.
	Strong affinity	Prohibits deployment to specified nodes. Ideal for preventing co-location with high-load services, avoiding fault-prone nodes, and isolating sensitive services from non-sensitive services.

In the **Add Node** list, select the nodes that meet the preceding configuration rules.

When you choose a pre-warmed model, only the pre-warmed nodes appear on the affinity scheduling page. Unwarmed nodes do not show. A message appears stating: The selected model is pre-warmed and will deploy automatically on the best node. Specifying a node might cause warmup to fail.

1.8.3 Traffic Policies

Overview

ModelArts real-time inference services provide multi-deployment traffic scheduling capabilities. For multiple deployment instances created under the same real-time inference service, an independent traffic weight value can be configured for each deployment. The platform automatically distributes and routes inference service call requests proportionally based on the relative weight of each deployment. This enables fine-grained O&M control, including model gray releases, A/B testing, failover traffic switching, and on-demand traffic allocation.

For real-time services with existing deployment configurations, you can adjust the traffic weights of each deployment under the current service by modifying the traffic weight settings. The specific rules are as follows:

- Single deployment traffic weight: The traffic weight value for a single deployment must be an integer not exceeding 100.
- Multi-deployment traffic weight allocation: Traffic is routed to different deployments proportionally.
- All deployment traffic weights set to 0: The system automatically distributes traffic evenly across all normally running deployments to ensure service continuity.
- Enable traffic mirroring: The system replicates a fixed percentage (10%) of live traffic to the corresponding deployment for shadow verification, without affecting user requests.

Table 1-42 Use cases

Use Case	Service Objective	Configuration Method	Value Delivered
Model gray release	Launches a new model version without interrupting services, minimizing failure risks through gradual rollout and validation.	1. Deploy a stable old version (V1) and a new version (V2) under the same real-time service. 2. Traffic weights: V1 = 90, V2 = 10. 3. Monitor QPS, latency, accuracy, and error rates, and gradually adjust to V1 = 50, V2 = 50. 4. If anomalies occur, switch V1 back to 100 for rapid rollback.	Zero-downtime release; controllable risk; completely seamless to users.

Use Case	Service Objective	Configuration Method	Value Delivered
Online A/B testing	Compares the real-world performance of multiple model versions or strategies to back iterative decisions with data.	1. Deploy V1 (baseline) and V2 (experimental group) under the same real-time service. 2. Traffic weights: V1 = 50, V2 = 50. 3. Utilize a unified entry to split traffic by weight while monitoring QPS, latency, accuracy, and error rates.	Leverages live production traffic for objective evaluation, eliminating offline evaluation bias.
Traffic mirroring (shadow testing)	Validates the compatibility, performance, and stability of a new version using real traffic without impacting users.	1. Enable traffic mirroring for the new version deployment. 2. The system automatically replicates 10% of production traffic to the new deployment. 3. The new deployment generates no real-world responses and is used purely for observation.	Real-world validation with zero service risk; ideal for pre-launch verification of LLMs or critical services.
Multi-deployment load balancing & auto scaling	Absorbs peak traffic spikes while saving resources during off-peak hours to maximize utilization and stability.	• Peak hours: Distribute weights equally across multiple deployments (e.g., 30:30:30) to handle the load together. • Off-peak hours: Retain one primary deployment at 100% weight, while scaling down or stopping others. • Setting all weights to 0: The system automatically balances traffic evenly to prevent single-point overload.	Dynamically adapts to varying workloads, eliminates resource waste, and enhances service availability.
Multi-model & multi-service traffic scheduling	Shares infrastructure resources under a single inference endpoint, allocating traffic based on priority or importance.	Core service model deployment: Weight = 70. Secondary/Experimental model deployment: Weight = 30. During resource constraints, lower non-core weights to safeguard core SLAs.	Promotes resource reuse, optimizes infrastructure costs, and establishes clear service priorities.

Use Case	Service Objective	Configuration Method	Value Delivered
Fast fault isolation & lossless rollback	Mitigates losses immediately when a specific model version experiences anomalies, keeping the overall service safe.	Instantly set the weight of the anomalous deployment to 0. Route 100% of the traffic back to the stable deployment version (Weight = 100). Isolate the problematic deployment for troubleshooting and patching.	Rapid loss mitigation; near-zero service impact; high O&M efficiency.
Rapid iteration & validation in test environments	Enables parallel development and testing validation without affecting production traffic.	Production deployment: Weight = 100. Test deployment: Weight unconfigured + traffic mirroring enabled. Validate functionality using shadow traffic before initiating a gradual gray release.	Strong isolation between production and testing; secure and highly efficient iteration cycles.

Constraints

Mirrored traffic occurs only in synchronous calls, not in asynchronous ones.

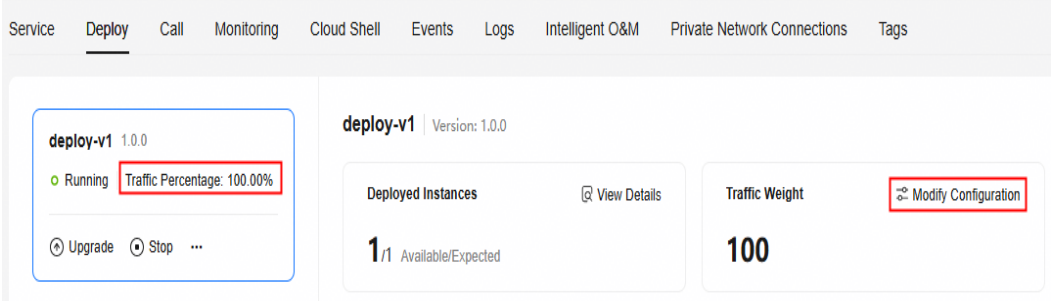
Viewing/Modifying Traffic Weights

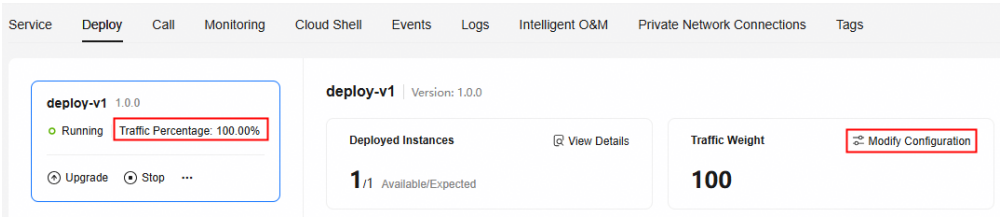
Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.

Method 1:

In the service list, click a service name to enter the service details page. In the **Deploy** tab of the service details page, you can view the actual traffic percentage of the service. The actual traffic percentage is calculated based on the traffic weight configuration and the deployment status.

Figure 1-68 Viewing the traffic percentage





In the traffic weight area, click **Modify Configuration** to modify the traffic percentage of the deployment.

Figure 1-69 Modify Traffic Weight

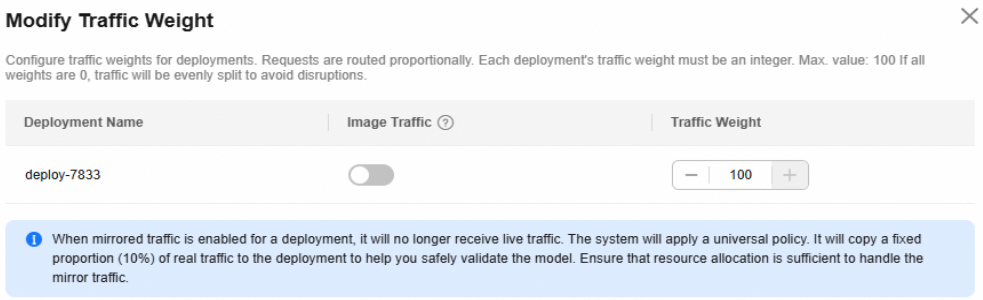


Table 1-43 Traffic weight policy parameters

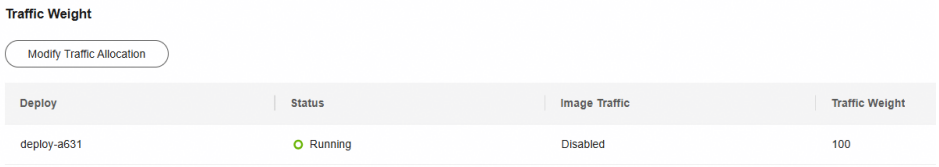
Parameter	Description
Mirrored Traffic	Also called shadow traffic. It is a secure deployment verification function that routes real-time copies of live traffic to a new version, without affecting actual user requests. It provides you with a shadow test environment that is identical to the production environment. Traffic mirroring requires that the service protocol be HTTP/HTTPS. When mirrored traffic is enabled for a deployment, it will no longer receive live traffic. The traffic weight is not used to estimate the traffic percentage. Instead, the system copies 10% of the actual traffic to the deployment using a standard policy. This helps you safely test the model. Ensure that resource quotas are sufficient to handle the mirrored traffic load. Mirrored traffic occurs in synchronous calls, not in asynchronous ones.
Traffic Weight	Configure traffic weights for deployments. Requests are routed proportionally. Each deployment's traffic weight must be an integer. Max. value: 100 If all weights are 0, traffic will be evenly split to avoid disruptions. Mirror traffic is not included in weight calculations for actual traffic. Its percentage is fixed by default and cannot be modified. Keep at least one deployment that receives live requests for each service.

Parameter	Description
Estimated Traffic Percentage	The estimated traffic percentage is the ratio of the traffic expected to be received by a single deployment instance group to the total traffic. The actual traffic percentage is calculated based on the traffic weight configuration and the deployment status.

Method 2:

In the service list, click a service name to enter the service details page. In the **Service** tab of the service details page, you can view the traffic weight of the service. Click **Modify Traffic Allocation** to modify the traffic weight of the deployment under the current service.

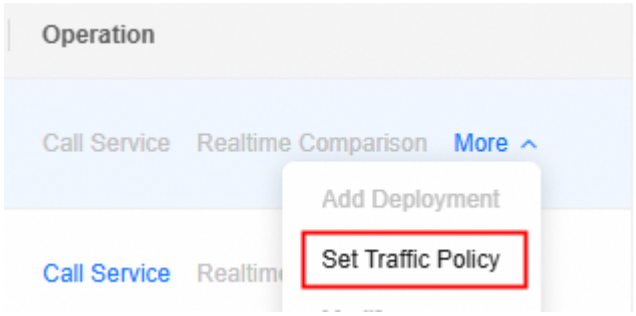
Figure 1-70 Viewing traffic weights



Method 3:

In the **Operation** column of the service list page, choose **More** > **Set Traffic Policy** to modify the traffic weight.

Figure 1-71 Set Traffic Policy



Click **OK**.

1.8.4 Storage Mounting

Overview

In ModelArts inference deployment scenarios, file storage mounting is a core capability designed to address LLM storage, data sharing, and loading acceleration. It adapts to high-performance, high-stability, and high-throughput inference service requirements. The core applicable scenarios are as follows:

1. Storage and loading of ultra-large models: LLMs and multimodal models can range in size from tens of gigabytes to terabytes, making it impossible to embed them directly into container images. Model files must be centrally stored using mounted storage so that inference instances can read them directly, preventing bloated images and deployment timeouts.
2. Data sharing across multiple instances: Multiple replicas of the same inference service, or different inference services altogether, often need to share configuration files, inference dependency data, and intermediate results. Mounted storage enables centralized data management and real-time synchronization, eliminating the need for repeated data copying.
3. Inference acceleration and rapid recovery: By utilizing local storage acceleration features, models are cached onto the host machine. When a container restarts due to a failure, it reuses the cached model, eliminating the time consumed by repeatedly pulling the model. This significantly improves service recovery efficiency and fits high-availability service scenarios.
4. Rapid reuse of pre-warmed models: High-frequency models can be warmed up in the resource pool in advance. When deploying an inference service, the pre-warmed model is mounted directly without needing to be reloaded, shortening deployment time and accommodating high-frequency, rapid-launch inference demands.
5. Custom model deployment: Custom models that are not built directly into ModelArts can be integrated into inference services via storage mounting. This flexibly accommodates deployment scenarios for third-party or in-house models.

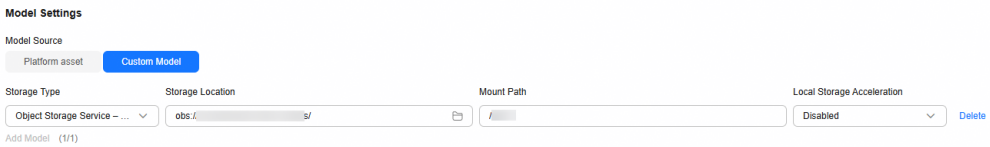
Storage Types

ModelArts real-time inference services support configuring the storage type within the model source, and it also supports mounting storage during unit configuration.

- Model source storage type

  The storage selected under the model settings on the deployment page is specifically dedicated to storing the model weight files required for running inference. Its primary purpose is to load the AI model, making it an absolute prerequisite for service startup. In most scenarios, it is read-only and will not be modified. For the specific configuration entry point, see **Configuring Deployment Settings**.

  Figure 1-72 Model Source > Storage Type

  Supported storage types for model sources: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models.

Table 1-44 Recommended storage types for model sources

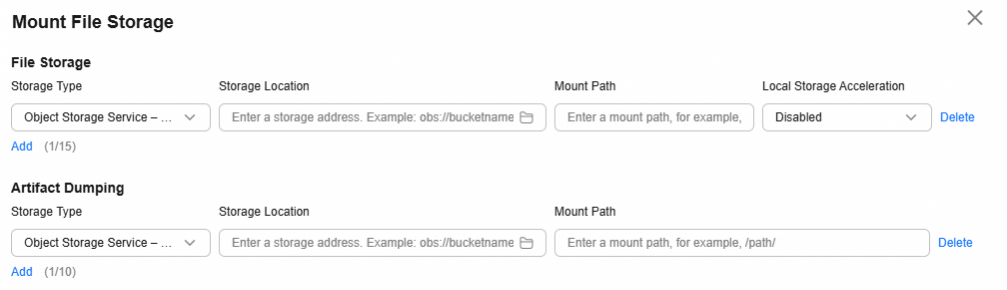
Storage Type	Use Case	Constraint
Object Storage Service – Bucket	• Small and medium-sized lightweight models (tens of MB to several GB). • Testing environments, demo scenarios, and non-core online services. • Low model update frequencies, low concurrency, and minimal access pressure. • Scenarios prioritizing the lowest storage cost over loading speed. • Simple inference tasks with no requirement for massive small file reading.	Not applicable to large or ultra-large models, high-concurrency inference, or scenarios requiring rapid batch service startup.
Object Storage Service – Parallel File System	• The preferred choice for popular LLM deployment (7B, 13B, 34B, and other general LLMs). • Multi-instance, high-concurrency inference, and clustered real-time inference. • Models with numerous files, deeply nested directories, or massive amounts of small files. • Interconnected training and inference data requiring frequent reads/writes of model dependency files. • Scenarios seeking a balance between high performance and low cost, avoiding premium-priced SFS Turbo. • Enterprise-grade, official online inference services.	/

Storage Type	Use Case	Constraint
Scalable File Service Turbo (SFS Turbo)	• Official production launch of ultra-large parameter models (100B+ parameters). • High-stability, low-latency, strong-consistency core services such as finance and government affairs. • Scenarios requiring second-level model loading and ultra-fast service restart/recovery. • Real-time modification of model configurations or dynamic reading and writing of weights during inference. • Extremely high requirements for I/O latency, stability, and fault recovery speed.	• Higher usage costs. • Can only be used in dedicated resource pools, and the dedicated resource pool must be associated with SFS Turbo.
Pre-warmed Model	• Rapid testing, temporary launches, and emergent scaling. • Frequent and repeated deployment of standardized general models. • Scenarios prioritizing the fastest startup speed without waiting for model downloads and loading. • Large-scale batch deployment of inference services within the same dedicated resource pool.	• Can only be used in dedicated resource pools. • Models must be warmed up on the nodes in advance. • Services can only be scheduled to pre-warmed nodes; the service cannot start if scheduling fails.

- **Unit Settings** > **Mount File Storage**

  This extra storage mounted within the inference unit does not store the primary model. Instead, it is dedicated to mounting service-operational supporting data, input/output files, and inference output dumps. Its purpose is to support the model throughout its entire service workflow. Classified as a supplementary service resource, it is optional and operates mostly in read/write mode. For the specific configuration entry point, see **Configuring Deployment Settings**.

Figure 1-73 Unit Settings > Mount File Storage



Supported storage types for file storage in unit settings: OBS buckets, OBS parallel file systems, SFS Turbo, and pre-warmed models.

Supported storage types for artifact dump in unit settings: OBS parallel file systems.

Object Storage Service – Bucket

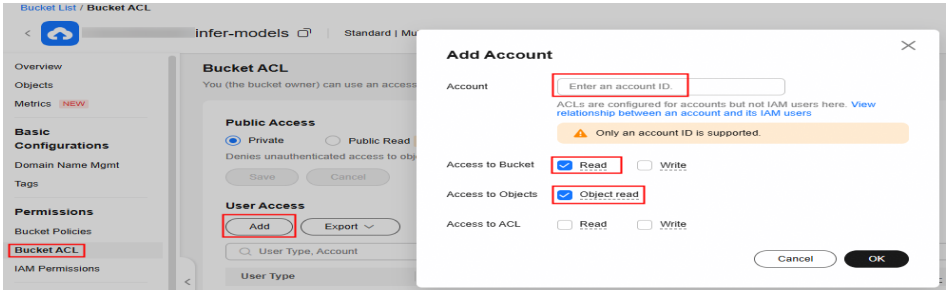
- **Storage Location**: Select an OBS bucket path. Cross-region OBS buckets cannot be selected. You can add up to 15 paths. Encrypt sensitive data before saving it to your OBS bucket.

  It is good practice to create a directory. Avoid using inventory directories or system directories with strict permissions.

  You can choose your own OBS bucket or enter a path. The path must start with **obs://** and end with a slash (/), like this: **obs://bucketname/path/**. For shared buckets from other users, you must enter the path.

  When ModelArts connects with IAM and you use a shared bucket path, the bucket owner must grant you access and read permissions in the bucket's ACL policy.

  Figure 1-74 Bucket ACL permissions

  The owner of the shared bucket must set up an agency in ModelArts to grant OBS permissions to all users. To do this, select all users for authorization and choose OBS for the function permission. For details, see **Configuring Agency Authorization for ModelArts with One Click**.

- **Mount Path**: Enter the container mount path, for example, **/obs-mount/**. It is good practice to create a directory. Avoid using inventory directories or system directories with strict permissions. Avoid using nested directories when configuring multiple mount paths.

- **Local Storage Acceleration**: Model data is pulled from external storage to the host (usually a CCE cluster node), and the model path on the host is mounted to the container directory specified by the user. If the local cache is still on the current node after the service container is restarted due to a fault, the model pulling process can be skipped. With local storage acceleration, when a service container is restarted due to a fault, the local cache can be reused, implementing fast recovery. The local cache can only be retained if the node where the service container is located is not destroyed and the storage volume is not deleted. If the node is faulty or the storage volume is cleared, the cache becomes invalid, and the model data needs to be pulled again.

  When using local storage acceleration, make sure the data disk has over 1,024 GiB of space in the dedicated resource pool.

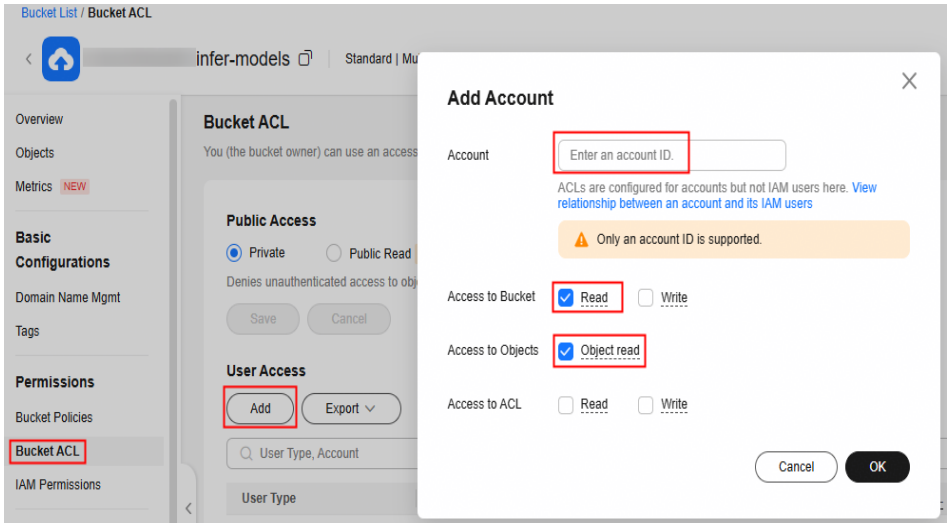
Object Storage Service – Parallel File System

- **Storage Location**: Select a storage path. A cross-region OBS parallel file system cannot be selected. Encrypt sensitive data before saving it to your OBS parallel file system.

  You can choose your own OBS parallel file system or enter a path. The path must start with **obs://** and end with a slash (/), like this: **obs://bucketname/path/**. For shared OBS paths from other users, you must enter the path.

  When ModelArts connects with IAM and you use a shared OBS path, the owner must grant you access and read permissions in the OBS object's ACL policy.

  Figure 1-75 ACL permissions



  The owner of the shared bucket must set up an agency in ModelArts to grant OBS permissions to all users. To do this, select all users for authorization and choose OBS for the function permission. For details, see **Configuring Agency Authorization for ModelArts with One Click**.

- **Mount Path**: Enter the container mount path, 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.

- Mount an empty directory to the container. If the directory is not empty, ensure that the directory does not contain any files that affect container startup. Otherwise, the files will be replaced, and the container cannot start normally. As a result, the workload may not be deployed.
- 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: _-.

- **Local Storage Acceleration**: Model data is pulled from external storage to the host (usually a CCE cluster node), and the model path on the host is mounted to the container directory specified by the user. If the local cache is still on the current node after the service container is restarted due to a fault, the model pulling process can be skipped. With local storage acceleration, when a service container is restarted due to a fault, the local cache can be reused, implementing fast recovery. The local cache can only be retained if the node where the service container is located is not destroyed and the storage volume is not deleted. If the node is faulty or the storage volume is cleared, the cache becomes invalid, and the model data needs to be pulled again.

  When using local storage acceleration, make sure the data disk has over 1,024 GiB of space in the dedicated resource pool.

Scalable File Service Turbo (SFS Turbo)

You can mount storage only if you use a dedicated resource pool for deploying the service. This pool must be associated with an SFS Turbo file system. For details, see **Associating the Network of a Dedicated Resource Pool File System with SFS Turbo**.

Parameters:

- **File System**/**File System Directory**: Select the corresponding SFS Turbo file system. A cross-region SFS Turbo file system cannot be selected.
- **Mount Path**: Enter the mount path of the container, for example, **/sfs-turbo-mount/**. Select a new directory. If an inventory directory is selected, the inventory files in it will be overwritten. Avoid using nested directories when configuring multiple mount paths.
- **Mount Mode**: **Read/Write** and **Read-only** are supported.
- **Local Storage Acceleration**: Model data is pulled from external storage to the host (usually a CCE cluster node), and the model path on the host is mounted to the container directory specified by the user. If the local cache is still on the current node after the service container is restarted due to a fault, the model pulling process can be skipped. With local storage acceleration, when a service container is restarted due to a fault, the local cache can be reused, implementing fast recovery. The local cache can only be retained if the node where the service container is located is not destroyed and the storage volume is not deleted. If the node is faulty or the storage volume is cleared, the cache becomes invalid, and the model data needs to be pulled again.

Notes:

- A file system can be mounted only once and to only one path. Each mount path must be unique. You can mount up to eight disks.
- If you need to mount multiple files, do not use the paths that are the same or similar, for example, **/obs-mount/** and **/obs-mount/tmp/**.
- Once you have chosen SFS Turbo, avoid disassociating SFS Turbo. Otherwise, mounting will not be possible. When you mount the backend OBS storage on the SFS Turbo page, make sure to set the client's umask permission to 777 for normal use.

Pre-warmed Model

- **Pre-warmed Model**: Select a model that has been warmed up in the ModelArts resource pool.
- **Mount Path**: Specify the mount path inside the container. You are advised to create a directory and do not select an existing directory or a system directory with strict permissions. The mount path must start and end with a slash (/). The path can contain at most 255 characters. The mount path must be the same as that of other model sources.

  NOTE

  If the task is assigned to nodes that are not pre-warmed, the deployment will fail due to resource limits.

1.8.5 Secret Mounting

Overview

In ModelArts inference service scenarios, secret mounting leverages the secret mechanism to achieve encrypted storage and secure injection of sensitive information. This prevents plaintext storage of sensitive data such as account passwords, AK/SK, database credentials, and certificates, thereby ensuring data security and privilege isolation during the runtime of the inference service.

Typical scenarios:

1. Model/Storage access authentication

   When an inference service needs to access resources like OBS or SFS Turbo, secret mounting injects the required AK/SK, access keys, or signature credentials. This avoids providing them in plaintext within configuration files or environment variables.

2. Third-party service integration

   When an inference service calls databases, message queues, API gateways, or authentication services, it mounts sensitive information such as database account passwords, API authentication keys, tokens, and private key certificates to secure cross-service communication.

3. Custom image execution

   When deploying an inference service based on a custom image, the application inside the image may need to read encrypted configurations, private keys, certificates, or license files. Secret mounting securely injects

these encrypted files into designated paths within the container, preventing sensitive information from being baked into the image.

4. Multi-tenant / multi-environment privilege isolation

   When multiple inference services share infrastructure under the same resource pool or cluster, secret mounting achieves service-level isolation of sensitive information. Each instance only retrieves its own required secrets, minimizing the attack surface for credential leaks.

Supported secret mounting methods in ModelArts

- Custom secret: Users manually enter sensitive or private data in key-value pairs, which are then encrypted and stored using the platform's native secret security mechanism. When deploying an inference service, these secrets can be directly mounted into the inference container as files for the service application to read and use. As a platform-native, lightweight secret management approach, it does not rely on external encryption services and stands as the most commonly used local private information configuration method for inference scenarios.
- **DEW**: Data Encryption Workshop (DEW) is a comprehensive cloud data encryption service, including Key Management Service (KMS), Key Pair Service (KPS), and Dedicated Hardware Security Module (Dedicated HSM). To use DEW to configure the key, go to the DEW console to **create a key**, select the DEW key on the ModelArts console, and enter the mount path.

Constraints

- Resource pool constraints: Secret mounting is supported for real-time inference services deployed in both dedicated resource pools and public resource pools.
- Mount method constraints: Only two mount methods are supported: Custom secret and DEW secret. Directly mounting plaintext sensitive information is not supported.
- Mount path constraints: The mount path inside the container must be a completely new directory and must not overwrite existing system or service directories. When mounting multiple secrets, paths must not overlap or cause parent-child directory conflicts (for example, **/secret/** and **/secret/custom/**). The path must start with a forward slash (/) and cannot exceed 255 characters in length.
- Permission association constraints: If **Associate Image User Group ID** is checked, secret mount permissions will be bound to the container image's startup user group; processes outside of this user group will be unable to read the secrets. Additionally, a mounted DEW secret must be unbound from the service before it can be deleted, otherwise it will cause instance anomalies or a failed restart.

Prerequisites

- Permissions: You must have activated permissions for ModelArts and DEW, including permissions for real-time inference service deployment, secret configuration, secret management, and Cloud Shell debugging. For details, see **Configuring Agency Authorization for ModelArts with One Click**.

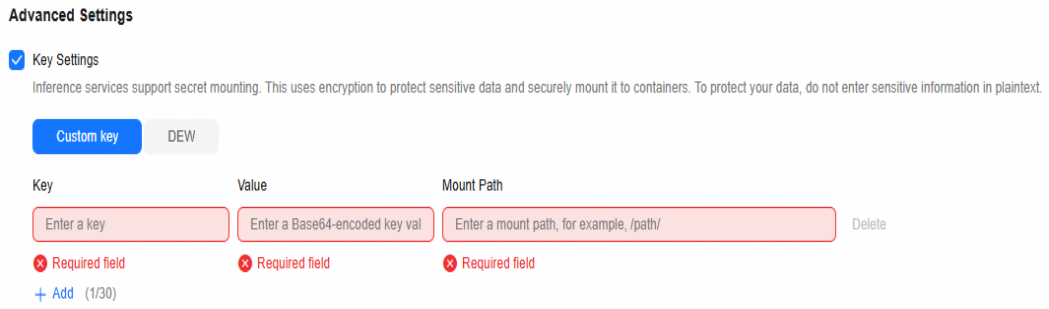
- Resource preparation: If you select a dedicated resource pool when deploying the inference service, ensure that the dedicated resource pool (supporting CPU, GPU, or NPU heterogeneous resource pools) has already been created.
- DEW preparation: For DEW mounting, log in to the DEW console and create a key-value pair secret.

Secret Mount Configuration

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, check **Key Settings** under **Advanced Settings** during the deployment configuration phase. For details, see **Deploying a Real-Time Inference Service Using a Single Node**.

Select either **Custom key** or **DEW**:

Figure 1-76 Key Settings > Custom key



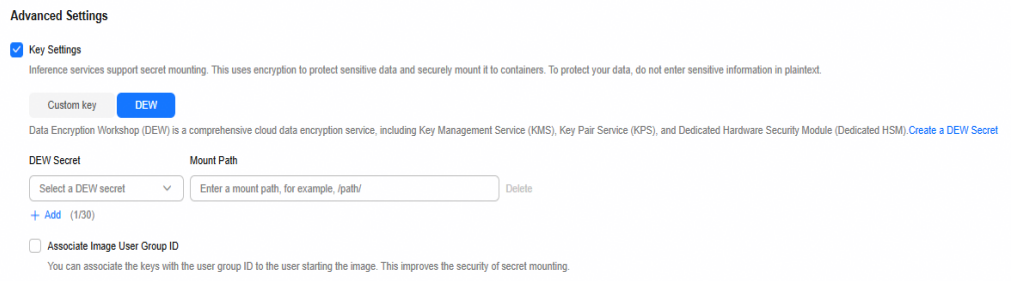
If you select **Custom key**, fill in the following fields:

- **Key**: A custom name, such as **infer-password**.
- **Value**: The Base64-encoded secret value.
- **Mount Path**: The path inside the inference container, such as **/secret/custom/**.

(Optional) If multiple secrets are required, click **Add**.

(Optional) Check **Associate Image User Group ID**. Once checked, secret mount permissions will be bound to the container image startup user's group ID; processes outside of this user group will be unable to read the secrets.

Figure 1-77 Key Settings > DEW



If you select **DEW**, fill in the following fields:

- **DEW Secret**: Select a secret from the drop-down list. If no options are available, you must navigate to the DEW console to **create a secret**.
- **Mount Path**: The path inside the inference container, such as **/secret/dew/**.

(Optional) If multiple secrets are required, click **Add**.

(Optional) Check **Associate Image User Group ID**. Once checked, secret mount permissions will be bound to the container image startup user's group ID; processes outside of this user group will be unable to read the secrets.

Configuration Verification

Go to the real-time inference service details page. On the Cloud Shell tab, select the deployment, deployment replica, and resource instance (pod). Once the connection is successful, check the mounted files.

The commands below are examples only. Replace the mount paths and file names with your actual configurations.

```
# View the custom secret
cat /secret/custom/infer-password
# View the DEW secret
cat /secret/dew/accessKeyId
```

If the contents can be read normally, the mount was successful.

Since public resource pools do not support Cloud Shell, the above verification method is only applicable to real-time inference services deployed in dedicated resource pools.

1.8.6 Intelligent Routing Policy

Overview

In large-scale real-time inference, the request distribution policy directly impacts response latency, load balancing, session continuity, and service stability. ModelArts real-time inference services provide intelligent routing policies that flexibly distribute traffic according to your service requirements. Typical scenarios:

- Load balancing and traffic dispersion

  In high-concurrency scenarios, requests must be evenly distributed across multiple instances to prevent single-instance overloading, reduce queuing delays, and maximize overall throughput.
- Low-latency prioritization

  For scenarios such as streaming generation and real-time Q&A, requests are preferentially routed to the instance with the lowest TTFT, significantly reducing response latency.
- IP affinity access

  Requests originating from the same client IP are consistently routed to the same instance, facilitating session caching, connection reuse, and log tracking.
- Service level objective (SLO) assurance

  Core services are given priority access to low-latency resources, ensuring that the latency targets of high-SLO priority services are met first.

- Dynamic load scheduling

  Requests are forwarded to the lightest-loaded instance based on a comprehensive evaluation of connections, TTFT, and custom metrics, thereby preventing resource waste.

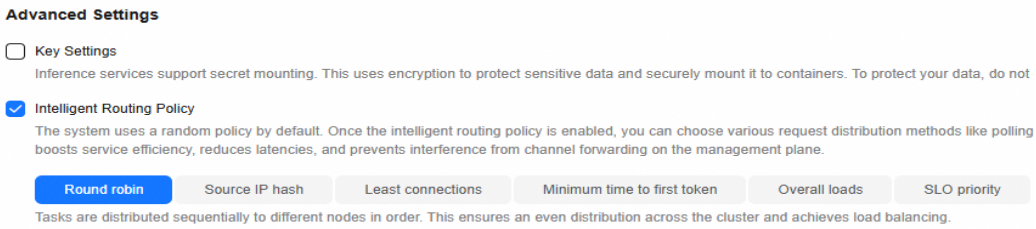
Constraints

- Resource pool limitations: Intelligent routing policies are only supported in dedicated resource pools; public resource pools are not supported.
- Policy combinations: A single deployment can only be configured with one intelligent routing policy; multiple policies cannot be superimposed.
- Instance requirements: Intelligent routing only takes effect on instances that are running and healthy. Faulty or anomalous instances are excluded from scheduling.

Configuring Intelligent Routing Policies

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, select **Intelligent Routing Policy** in **Advanced Settings** and configure the policy based on service requirements. For details about how to deploy a real-time service, see **Deploying a Real-Time Inference Service Using a Single Node**.

Figure 1-78 Intelligent Routing Policy



If intelligent routing is enabled, set intelligent routing policies. The following policies are supported:

Table 1-45 Intelligent routing policies

Policy	Core Rule	Use Case	Advantage	Not Applicable To
Round robin	Distributes requests sequentially to different instances to ensure traffic is evenly distributed across the cluster, achieving load balancing.	• General inference services: Image classification, text moderation, content recognition, data structuring, and other standardized inference tasks with single, independent calls and no contextual dependencies. • Uniform traffic stress testing: Service stress testing and cluster performance acceptance to evenly saturate the compute of all instances. • Basic general service clusters: Unified AI compute outlets for small and medium-sized enterprises with no special latency or session requirements.	Simplest configuration, lowest scheduling overhead, and complete elimination of single-instance traffic accumulation.	Multi-turn dialogues, long-connection interactions, and services requiring fixed-instance caching.

Policy	Core Rule	Use Case	Advantage	Not Applicable To
Source IP hash	Calculates a hash value based on the client's IP address to allocate requests, ensuring that requests from the same IP address are consistently routed to the same instance. Suitable for session caching scenarios.	• Fixed client access services: Enterprise intranet office AI tools and unified inference API calls by employees in fixed regions. • Local cache reuse scenarios: Instances cache popular inference results locally, allowing repeated requests from the same IP address to hit the cache directly, significantly boosting speed. • Access log traceability: Aggregates requests by client IP address, making it easier for O&M teams to troubleshoot user access anomalies and calculate regional call volumes. • Lightweight fixed session services: Simple interactive services that distinguish users based solely on access IP addresses without requiring account logins.	Inherently achieves simple session binding and features a high connection reuse rate.	Access of a large number of dynamic public IP addresses and mobile IP addresses

Policy	Core Rule	Use Case	Advantage	Not Applicable To
Least connections	Tracks the active connections of each inference instance in real time and automatically distributes new requests to the idle instance with the fewest connections to prevent overloading.	• Persistent-connection inference services: Streaming voice recognition, real-time video analysis, and long-running AI task processing. • Services with burst traffic fluctuations: E-commerce promotion review, content risk control during peak hours, and temporary batch inference tasks. • Inconsistent compute clusters: Clusters with mismatched performance configurations across instances, automatically bypassing high-load instances. • High-concurrency short-duration requests: Government bulk data processing and batch file AI parsing.	Rapidly digests idle compute, preventing certain instances from freezing due to full loads while others remain idle.	Extreme low-latency prioritization or fixed session interaction scenarios.

Policy	Core Rule	Use Case	Advantage	Not Applicable To
Minimum TTFT	Requests are routed to the node with the lowest average time to first token. This method aims to minimize the wait time between receiving a request and starting its processing. This latency applies even when system resources are available immediately.	• LLM real-time dialogue: Intelligent customer service, online Q&A, real-time copywriting generation, and instant human-machine interaction. • Frontend real-time display: Web-based AI writing, in-app smart Q&A, and real-time inference feedback in pop-ups. • High low-latency demands: Industrial real-time quality inspection, in-vehicle real-time AI analysis, and device edge-linked inference. • User-experience-first C-end services: Public-facing AI tools that prioritize perceived response speeds.	Minimizes first-response latency to the absolute limit, providing the optimal user experience.	Large-scale offline inference or throughput-focused services where latency is not a critical factor.

Policy	Core Rule	Use Case	Advantage	Not Applicable To
Overall loads	Requests are routed to the instance with the lowest overall pressure, considering factors such as the number of connections, time to first token, and custom metrics. It directs new requests to less busy instances to avoid resource waste and overloading. Overall loads require configuring the protocol (HTTPS or HTTP) and port number of the **Metric Collection API**. The port number must be an internal listening port of the image; choose a port number between 1024 and 65535. Recommended ports are 9090, 9100, and 8088.	• Mixed service clusters: A single cluster simultaneously handling dialogue inference, batch processing, and content moderation. • Unified AI inference platforms for large enterprises: Compute pools shared by multiple departments where service types are chaotic with no uniform call characteristics. • 24/7 stable O&M: Unattended automatic scheduling that fully balances resource pressure across all dimensions. • Heterogeneous compute hybrid clusters: Mixed deployment of different compute nodes (e.g., GPU and NPU) to intelligently match optimal resources.	Yields the highest global resource utilization and ensures the smoothest overall cluster operation.	Services with a single extreme performance demand or a strong binding to fixed sessions.

Policy	Core Rule	Use Case	Advantage	Not Applicable To
SLO priority	Pre-classifies different inference services into service level priorities (configurable from 0 to 3, where 0 is the highest) to preferentially guarantee the latency of high-priority services.	• Government/Enterprise tiered service scheduling: High priority for core government services and livelihood-critical AI services; low priority for general office assistant AI. • Enterprise core service protection: Revenue-related paid AI services are prioritized over internal free testing services. • Service rate-limiting tiers: During peak traffic periods, priority is given to paid customers and core channel calls, while secondary service traffic is restricted. • Cloud compute resource quota management: Allocates compute weights based on service importance to achieve resource priority isolation.	Strictly guarantees the SLA service metrics of core services, prioritizing their stability and availability.	Universal inference clusters where all services have equal rights and no hierarchical tiers exist.

1.9 Reliability

1.9.1 Graceful Shutdown of Real-Time Services

Overview

Graceful shutdown for real-time inference services is used in scenarios such as service discontinuation, upgrades, scale-ins, and fault migrations. Its purpose is to smoothly complete in-flight requests, release resources, save states, and avoid forced interruptions, thereby guaranteeing service continuity and data integrity.

Typical scenario:

- Normal service discontinuation: During service iterations, version replacements, or resource reclamation, the system waits for current inference requests to complete before terminating instances, preventing client-side errors.

- Rolling upgrades: When updating models or container images, old instances are gracefully shut down step-by-step while new instances are pulled up, achieving a smooth upgrade with zero disruption.
- Instance scale-in: When reducing the number of replicas or releasing idle resources, priority is given to gracefully terminating idle instances without affecting requests currently being processed.
- Node and hardware fault migration: When NPU, CPU, or switch anomalies trigger a migration, graceful shutdown saves the temporary inference context, reducing task failures.
- LLM and long-task inference: For streaming or long-text inference tasks that take a long time to execute, graceful shutdown prevents forced terminations that result in incomplete outputs or broken sessions.

Constraints

- Health check linkage: For services already configured with health checks, enabling graceful shutdown introduces a forced 3-minute delay before the service stops after receiving a stop command.
- Resource pool limitations: Both dedicated resource pools and public resource pools are supported.
- Deployment modes: Both the basic mode and the multi-role separation mode are supported. For the multi-role separation mode, the configuration must be applied uniformly across all inference units.
- Command dependencies: Shutdown commands rely on basic system utilities inside the container (such as **sh**, **kill**, and **trap**). If these dependencies are missing, the execution will fail.

Graceful Shutdown Configuration

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, select **Graceful Shutdown** in **Unit Settings** > **More Settings**, and set the shutdown time and command. For details, see **Deploying a Real-Time Inference Service Using a Single Node**.

Shutdown Timeout (s): 30s for common services; 60s to 300s for LLMs or long tasks.

Shutdown Command (executed inside the container): Choose and enter only one of the three examples below:

- Example 1 (terminating a process)
  ```
  kill -SIGTERM $(pgrep -f app.py)
  ```
  Application: Directly and gracefully terminates the primary service process.

  Principle: Sends a standard graceful termination signal, allowing the process to clean up and exit on its own.

  Scenarios: Python scripts, standalone Flask/FastAPI process services.
- Example 2 (graceful exit)
  ```
  trap 'exit 0' SIGTERM
  ```
  Application: Directly and gracefully terminates the primary service process.

  Principle: Sends a standard graceful termination signal, allowing the process to clean up and exit on its own.

Scenarios: Shell resident services, background daemon processes.

- Example 3 (releasing connections/saving state)
  ```
  /bin/sh -c stop_service.sh
  ```
  Application: Complex shutdown logic (multi-step cleanup).

  Principle: Calls a custom shutdown script inside the container to execute a complete suite of actions, such as saving logs, disconnecting, flushing cache to disk, and closing child processes.

  Scenarios: Services with complex service workflows that require custom cleanup operations.

After the configuration is complete, stop or upgrade the service, and observe that in-flight requests finish normally without errors.

1.9.2 Rolling Upgrades for Real-Time Service Deployment

Overview

Rolling upgrade is a core capability of ModelArts real-time inference services that enables smooth iteration and zero-disruption updates. By gradually replacing deployment replicas, it allows for service version upgrades, image updates, model replacements, and configuration changes, keeping the entire process transparent to the services and ensuring requests are never interrupted.

Typical scenario:

- Model version iteration: Upgrading large or small models after updating weights or fine-tuning, preventing a complete service outage.
- Image updates: Patching vulnerabilities, optimizing performance, or upgrading dependency packages by replacing instances with zero downtime.
- Configuration changes: Adjusting resource specifications, environment variables, or health check parameters with smooth effectuation.
- Gray releases: Launching new versions in batches and rolling them out completely only after validating stability, thereby minimizing risk.
- High-availability O&M: Rolling restarts of instances when service anomalies occur, ensuring service continuity.

Constraints

- Resource pool constraints: Both dedicated resource pools and public resource pools support rolling upgrades.
- Image constraints: Custom images and preset images are both supported; however, the image must be able to start normally without fatal errors.
- Replica count constraints: True zero-disruption upgrade can only be achieved when the unavailable replica count ≥ 0. If the unavailable replica count reaches 100%, a brief interruption will occur.
- Health check constraints: You are advised to configure health checks beforehand to prevent new instances from receiving traffic before they are fully ready.
- Upgrade interruption constraints: If insufficient resources or image errors are encountered during the upgrade process, the upgrade will be terminated, and the old instances will continue to run.

Configuring a Rolling Upgrade

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, set **Max Surge Replicas** and **Max Unavailable Replicas** under **Deployment Management Settings** > **More Settings**. For details, see **Deploying a Real-Time Inference Service Using a Single Node**.

Max Surge Replicas: The maximum percentage by which the number of deployment replicas can exceed the target replica count during each rolling upgrade step.

Max Unavailable Replicas: The maximum percentage by which the number of deployment replicas can fall below the target replica count during each rolling upgrade step. If **Max Unavailable Replicas** equals the target replica count, there is a risk of downtime (**Min Available Replicas** = **Total Replicas** – **Max Unavailable Replicas**).

Table 1-46 Configuration suggestions

Use Case	Configuration Suggestion
General production configuration (zero disruption)	Max Surge Replicas: 1% Max Unavailable Replicas: 1%
Fast upgrade configuration (low latency)	Max Surge Replicas: 25% Max Unavailable Replicas: 25%
Gray testing configuration (small batch)	Max Surge Replicas: 10% Max Unavailable Replicas: 10%
Single-instance emergency configuration (brief interruption allowed)	Max Surge Replicas: 100% Max Unavailable Replicas: 100%

1.9.3 Real-Time Service Health Check

Overview

ModelArts real-time service health check is a core capability that ensures the stable running and high availability of AI inference services. It periodically checks the startup, readiness, and running status of service instances, automatically identifies abnormal instances, and triggers repair or isolation operations. This prevents issues such as invalid traffic forwarding, service freezing, and process breakdown. It is applicable to the following core scenarios:

- Slow model loading: When deploying foundation models (such as 100-B LLMs or multimodal models), the initial loading process takes a long time. Health checks prevent the "false active" dilemma, where the service status shows as "Running" but cannot actually respond to incoming requests.

- Precise traffic distribution: This ensures that inference requests are forwarded exclusively to instances that are fully ready and capable of processing tasks normally, successfully eliminating request failures and timeouts.
- Automated fault self-healing: When an anomaly occurs, such as an instance process crashing, resource exhaustion, or missing dependencies, the platform automatically restarts the instance without manual intervention, thereby boosting service availability.
- Rolling upgrade assurance: During a service upgrade, health checks are used to verify the availability of new instances, enabling zero-downtime, disruption-free upgrades.
- Custom image deployment: For inference services developed based on custom container images, health checks verify whether the internal service processes within the image are running as expected.

Constraints

- API/Script constraints: Before configuring a health check, you must deploy the health check API (HTTP/HTTPS) or probe script inside the image in advance; otherwise, it will directly cause the model deployment to fail.
- Stop delay: For services with health checks configured, the stop command is delayed by 3 minutes to prevent request interruption.
- Resource pool constraints: Health checks can be configured for both public and dedicated resource pools.

Prerequisites

The health check API (default path: **/health**) or detection script (for example, **/home/ma-user/health.sh**) has been deployed in the image.

Health Check Configuration

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, select **Health Check** in the **Unit Settings**. For details, see **Deploying a Real-Time Inference Service Using a Single Node**. The following describes health check parameters.

Health checks support three types of probes. Choose one or more based on your needs. Probes can be set up alone or together. If you configure a startup probe, other probes will not work.

- Startup probe: This probe checks if the instance has started. If a startup probe is configured, the liveness or readiness probe is not executed until the startup probe is successful, allowing sufficient time for the application to complete initialization. If the startup probe fails, the instance is restarted. If startup probe is not configured, the service status changes to success immediately after the service is scheduled. The service may be in the **Running** state and the prediction cannot be performed because the model is being loaded. It is recommended for foundation models and services that take a long time to start.
- Readiness probe: This probe verifies whether the instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be

routed to the instance until the probe succeeds. It is recommended for scenarios sensitive to traffic distribution.

- Liveness probe: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. It is recommended for high availability and fault self-healing scenarios.

Table 1-47 Startup probe parameters

Parameter	Description	Example
Check Method	Select the health check method. Select either of the following options, which match the health check capability of the custom image: • **HTTP**: Applies to containers offering HTTP/HTTPS services. The cluster regularly sends HTTP/HTTPS GET requests to these containers. A response code between 200 and 399 means the check passes. • **Command**: The container must have an executable command (a detection script). The cluster runs this command regularly. A return value of 0 means the check passed.	**HTTP**
URL	This parameter appears when you select **HTTP**. It sets the path for the health check to send an HTTP GET request. Make sure your application listens on this path and responds with a 200-399 status code to indicate success. The path must match the pattern ^[/][a-zA-Z0-9-_/:]{0,1024}.	**/health**
Command	This parameter appears when you choose **Command**. Enter the command to check the container's health status. Before using this method, you must package the required programs and tools in the container image. The cluster executes commands directly in the container. Host file systems and other containers' file systems are inaccessible. If the dependent programs or tools (such as curl, nc, or custom scripts) are not included in the image, error message "Command not found" will be displayed. If a shell script is executed, you must specify a script interpreter. The cluster does not provide an interactive terminal, so you cannot execute scripts directly. You must use the interpreter to invoke the script. For example, if the script is located in **/home/ma-user/health.sh**, specify **sh /home/ma-user/health.sh** when running the command.	/

Parameter	Description	Example
Interval (s)	Interval for performing a health check, in seconds. This value sets how often the probe does health checks. Shorter intervals find problems faster but use more resources. Longer intervals save resources but might delay problem detection. The value ranges from **1** to **2147483647**.	**10**
Delay (s)	Number of seconds to wait after the container is started before starting the probe. The value ranges from **1** to **2147483647**. If a startup probe is set, the liveness and readiness probes start their delay only after the startup probe succeeds. If the health check interval is longer than the delay, the delay is skipped.	**60**
Timeout (s)	Number of seconds to wait after the check request times out. If no response is received within this time, the check fails. For applications that process slowly or have poor network conditions, increase this value to prevent incorrect failure judgments. The value ranges from **1** to **2147483647**.	**30**
Failure Threshold	Number of consecutive health check failures that mark a container as unhealthy. This mechanism stops brief network issues or temporary application overload from causing false alarms. For example, a default setting of 12 means a restart or traffic removal happens only after 12 straight health check failures. The value ranges from **1** to **2147483647**. If the service has too many failed health checks in a row during startup, it goes into an abnormal state. If the service has too many failed health checks in a row during running, it goes into an alarm state.	**1800**
Protocol	Network protocol used for health checks. The value must be the same as that of the real-time inference service. • HTTP: HTTP is used for detection. • HTTPS: HTTPS is used for detection. This protocol is suitable for scenarios that require encrypted transmission.	**HTTP**

Table 1-48 Readiness probe parameters

Parameter	Description	Example
Check Method	Select the health check method. Select either of the following options, which match the health check capability of the custom image: • **HTTP**: Applies to containers offering HTTP/HTTPS services. The cluster regularly sends HTTP/HTTPS GET requests to these containers. A response code between 200 and 399 means the check passes. • **Command**: The container must have an executable command. The cluster runs this command regularly. A return value of 0 means the check passed.	**HTTP**
URL	This parameter appears when you select **HTTP**. It sets the path for the health check to send an HTTP GET request. Make sure your application listens on this path and responds with a 200-399 status code to indicate success. The path must match the pattern ^[/][a-zA-Z0-9-_/:]{0,1024}.	**/health**
Command	This parameter appears when you choose **Command**. Enter the command to check the container's health status. Before using this method, you must package the required programs and tools in the container image. The cluster executes commands directly in the container. Host file systems and other containers' file systems are inaccessible. If the dependent programs or tools (such as curl, nc, or custom scripts) are not included in the image, error message "Command not found" will be displayed. If a shell script is executed, you must specify a script interpreter. The cluster does not provide an interactive terminal, so you cannot execute scripts directly. You must use the interpreter to invoke the script. For example, if the script is located in **/home/ma-user/health.sh**, specify **sh /home/ma-user/health.sh** when running the command.	/
Interval (s)	Interval for performing a health check, in seconds. This value sets how often the probe does health checks. Shorter intervals find problems faster but use more resources. Longer intervals save resources but might delay problem detection. The value ranges from **1** to **2147483647**.	**10**

Parameter	Description	Example
Delay (s)	Number of seconds to wait after the container is started before starting the probe. The value ranges from **1** to **2147483647**. If a startup probe is set, the readiness probe starts its delay only after the startup probe succeeds. If the health check interval is longer than the delay, the delay is skipped.	**60**
Timeout (s)	Number of seconds to wait after the check request times out. If no response is received within this time, the check fails. For applications that process slowly or have poor network conditions, increase this value to prevent incorrect failure judgments. The value ranges from **1** to **2147483647**.	**30**
Failure Threshold	Number of consecutive health check failures that mark a container as unhealthy. This mechanism stops brief network issues or temporary application overload from causing false alarms. For example, a default setting of 12 means a restart or traffic removal happens only after 12 straight health check failures. The value ranges from **1** to **2147483647**. If the service has too many failed health checks in a row during startup, it goes into an abnormal state. If the service has too many failed health checks in a row during running, it goes into an alarm state.	**3**
Protocol	Network protocol used for health checks. The value must be the same as that of the real-time inference service. ● HTTP: HTTP is used for detection. ● HTTPS: HTTPS is used for detection. This protocol is suitable for scenarios that require encrypted transmission.	**HTTP**

Table 1-49 Liveness probe parameters

Parameter	Description	Example
Check Method	Select the health check method. Select either of the following options, which match the health check capability of the custom image: • **HTTP**: Applies to containers offering HTTP/HTTPS services. The cluster regularly sends HTTP/HTTPS GET requests to these containers. A response code between 200 and 399 means the check passes. • **Command**: The container must have an executable command. The cluster runs this command regularly. A return value of 0 means the check passed.	**HTTP**
URL	This parameter appears when you select **HTTP**. It sets the path for the health check to send an HTTP GET request. Make sure your application listens on this path and responds with a 200-399 status code to indicate success. The path must match the pattern ^[/][a-zA-Z0-9-_/:]{0,1024}.	**/health**
Command	This parameter appears when you choose **Command**. Enter the command to check the container's health status. Before using this method, you must package the required programs and tools in the container image. The cluster executes commands directly in the container. Host file systems and other containers' file systems are inaccessible. If the dependent programs or tools (such as curl, nc, or custom scripts) are not included in the image, error message "Command not found" will be displayed. If a shell script is executed, you must specify a script interpreter. The cluster does not provide an interactive terminal, so you cannot execute scripts directly. You must use the interpreter to invoke the script. For example, if the script is located in **/home/ma-user/health.sh**, specify **sh /home/ma-user/health.sh** when running the command.	/

Parameter	Description	Example
Interval (s)	Interval for performing a health check, in seconds. This value sets how often the probe does health checks. Shorter intervals find problems faster but use more resources. Longer intervals save resources but might delay problem detection. **Set the liveness probe check interval long enough to avoid premature container restarts.** The value ranges from **1** to **2147483647**.	**10**
Delay (s)	Number of seconds to wait after the container is started before starting the probe. The value ranges from **1** to **2147483647**. If a startup probe is set, the liveness probe starts its delay only after the startup probe succeeds. If the health check interval is longer than the delay, the delay is skipped.	**60**
Timeout (s)	Number of seconds to wait after the check request times out. If no response is received within this time, the check fails. For applications that process slowly or have poor network conditions, increase this value to prevent incorrect failure judgments. The value ranges from **1** to **2147483647**.	**30**
Failure Threshold	Number of consecutive health check failures that mark a container as unhealthy. This mechanism stops brief network issues or temporary application overload from causing false alarms. For example, a default setting of 12 means a restart or traffic removal happens only after 12 straight health check failures. The value ranges from **1** to **2147483647**. If the service has too many failed health checks in a row during startup, it goes into an abnormal state. If the service has too many failed health checks in a row during running, it goes into an alarm state.	**6**
Protocol	Network protocol used for health checks. The value must be the same as that of the real-time inference service. • HTTP: HTTP is used for detection. • HTTPS: HTTPS is used for detection. This protocol is suitable for scenarios that require encrypted transmission.	**HTTP**

After deploying the real-time inference service, check the health logs and exception records in the monitoring tab of the service details page.

1.9.4 Auto Restart upon a Real-Time Service Fault

Scenario

If a fault occurs on an NPU, switch, or hardware, the system automatically moves affected services to another node, speeding up recovery for real-time services.

Constraints

Some capabilities are only supported by Snt9b and Snt9b2 resources, such as NPU fault recovery.

Enabling Auto Restart

You can enable auto restart when configuring the deployment information of a real-time service.

Some capabilities are only supported by Snt9b and Snt9b2 resources.

1.9.5 Auto Rebuild upon a Real-Time Service Fault

Overview

Automatic rebuild is a core capability that guarantees high availability, self-healing, and stable operation for real-time inference services. It is applicable to the following scenarios:

- Accidental container or process crashes, requiring fast recovery

  When an inference instance (pod) exits unexpectedly due to process anomalies, out of memory (OOM) errors, missing dependencies, or code bugs, the platform automatically reconstructs the pod and restarts the inference service without manual intervention, minimizing service downtime.
- Smooth application of deployment configuration changes

  After modifying configurations such as images, environment variables, resource specifications, or mount paths, automatic reconstruction can automatically restart and rebuild instances based on predefined policies, allowing new configurations to take effect quickly while ensuring service continuity.
- Fault tolerance during rolling upgrades and version iterations

  When upgrading services, switching model versions, or changing deployment configurations, if some instances fail to upgrade, automatic reconstruction can automatically restore the abnormal instances to prevent the overall service from being affected.
- Fast migration and reconstruction after underlying node failures

  In the event of dedicated resource pool node crashes, hardware anomalies, or NPU failures, automatic rebuild can schedule pods to healthy nodes and rebuild the instances, guaranteeing the continuous availability of inference services.

- Long-term stable operation and unattended production environments

  For 24/7 real-time services such as LLM real-time inference, intelligent customer service, and real-time prediction, the system relies on automatic rebuild capabilities to achieve fault self-healing, lower O&M costs, and reduce manual intervention.

When automatic rebuild is enabled, if a pod restarts due to deployment configuration changes or failures, the platform will automatically rebuild it using the selected policy. If disabled, the platform will not intervene.

Rebuild policies:

- **Deployment replica rebuild**: When a pod restarts, the entire deployment replica is rebuilt.
- **Unit rebuild**: When a pod restarts, the entire unit is rebuilt.
- **Unit replica rebuild**: When a pod restarts, the entire unit replica is rebuilt.
- **Pod rebuild**: When a pod restarts, the entire pod is rebuilt.

Table 1-50 Suggestions on configuring automatic rebuild policies

Use Case	Recommended Policy	Description
Single-PU/Small-model inference, no communication dependencies between pods	Pod rebuild	Minimizes the scope of fault impact. Only the faulty pod is reconstructed, leaving the remaining pods unaffected.
Multi-PU inference (TP), strong communication dependencies within the same unit	Unit replica rebuild	A failure in one pod means the entire unit replica becomes unavailable. All pods within that replica must be reconstructed to ensure communication consistency.
Multi-unit inference (PP+TP), sequential dependencies between units	Unit rebuild	Multiple pods within a unit jointly handle the computing tasks for a specific stage. A failure in any single pod will prevent that stage from completing its computation. The entire unit must be reconstructed to restore computing capability and guarantee RankTable consistency across units.
Scenarios requiring strong global state consistency (e.g., distributed inference)	Deployment replica rebuild	A failure in any single pod impacts the entire global system. All pods across the entire deployment replica must be reconstructed to ensure the global RankTable and HCCL are renegotiated.

Constraints

- Resource pool limitations: Both dedicated resource pools and public resource pools support automatic rebuild.
- Dependence on health checks: If health checks (readiness / liveness probes) are not configured, the platform cannot precisely identify anomalies, which may lead to invalid reconstructions or reconstruction delays.
- Difference between automatic rebuild and automatic restart: Automatic rebuild focuses on instance rebuild after configuration changes or software anomalies; automatic restart focuses on scheduling and restarting after hardware, NPU, or switch failures. The two can be used in combination.
- Under unit rebuild, unit replica rebuild, or pod rebuild policies: The system must wait for the container to exit before rebuild. Due to graceful shutdown mechanisms or batch execution, when the number of resource instances for a unit replica is greater than 1, the total time required for containers to exit may reach approximately twice the graceful shutdown duration.

Prerequisites

The real-time inference service is in the **Running** state.

Enabling Auto Rebuild

Method 1:

When adding a deployment, choose **Unit Settings** > **More Settings**, select **Automatic Rebuild**, and select a rebuild policy.

Method 2:

If automatic rebuild is not enabled during deployment, you can update the deployment service and change the configurations. The procedure is as follows:

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the name of the target real-time service to enter its details page. On the details page, switch to the **Deploy** tab, select the target deployment card, and click **Upgrade**.
3. Under **Unit Settings** > **More Settings**, select **Automatic Rebuild**, and select a rebuild policy (one out of four options).
   - **Pod rebuild**: Only the abnormal pod is rebuilt.
   - **Unit rebuild**: The entire inference unit is rebuilt.
   - **Unit replica rebuild**: When a pod restarts, the entire unit replica is rebuilt.
   - **Deployment replica rebuild**: The entire deployment replica is rebuilt.
4. Save the configuration to enable automatic rebuild.

   After this function is enabled, if a pod exits due to an exception or restarts due to configuration changes, the platform will automatically rebuild the instance based on the selected policy.

Related Operations

You are advised to configure health checks at the same time to improve anomaly detection accuracy and reduce invalid rebuild. For details, see **Real-Time Service Health Check**.

1.9.6 Real-Time Service Intelligent O&M (HRA Plugin)

Overview

The core purpose of intelligent O&M (HRA plugin) is to resolve inappropriate P/D allocation ratios in LLM deployments. It adapts to highly complex scenarios such as large-scale inference, cross-node deployment, and load balancing. Typical scenarios are as follows:

- **High-concurrency LLM inference scenarios**

  LLM inference is divided into prefill units and decode units. During service peaks, the P/D ratio can easily become imbalanced, leading to low resource utilization, high response latency, and insufficient throughput. Intelligent O&M uses simulation algorithms to provide optimal ratio recommendations. Adjusting these manually improves resource utilization, reduces latency, and increases service throughput.

- **Performance optimization with fixed resource capacity**

  When resource pool hardware specifications (GPU/CPU/memory) are fixed and cannot be scaled out, Intelligent O&M recommends the optimal P/D ratio to maximize inference performance within existing resource limits, preventing resource waste or performance bottlenecks.

- **Iterative tuning for multi-version models**

  When multiple deployments and model versions run concurrently under the same service, different models have varying P/D ratio requirements. Intelligent O&M can provide exclusive ratio recommendations based on the asset tags of each model, adapting to the compute demands of different models and ensuring the stable operation of multi-version services.

- **Fine-grained resource management**

  This applies when inference instance resource allocation must be precisely controlled to avoid cost increases from over-provisioning or service anomalies from under-provisioning. Intelligent O&M provides real-time ratio recommendations to replace traditional manual empirical configurations, achieving fine-grained and scientific resource scheduling.

Constraints

- Resource pool limitations: The intelligent O&M feature is only supported in physical dedicated resource pools that have the HRA plugin installed. Public resource pools and dedicated resource pools without the HRA plugin do not support this capability.
- Model asset limitations: Model assets deployed for real-time services must carry the "dynamic ratio recommendation" tag. Models without this tag cannot trigger intelligent O&M monitoring and recommendations. Currently, only model assets pre-built on ModelArts support this feature; custom models are not supported.

- Functional dependency limitations: Intelligent O&M is a manual, assisted tuning capability and does not depend on auto-scaling features. You must manually adjust the P/D ratio based on the recommended values; automatic adjustment of instance ratios is not supported.
- Scope of application limitations: This feature only adapts to inference services using LLM cross-node deployments and disaggregated P/D architectures. Simple small models and single-node deployment services do not have the concept of a P/D ratio and cannot use this feature.
- Unit name limitations: The current algorithm only supports monitoring the ratio of prefill and decode inference units, and these names serve as the unique basis for metric collection. Do not modify these unit names; otherwise, data collection anomalies will occur.

Prerequisites

- The real-time service has been deployed in a physical dedicated resource pool with the HRA plugin installed, and the service status is **Running**. To check whether a dedicated pool has the HRA plugin installed, see **Viewing Resource Pool Plug-ins**.
- The deployed model asset carries the "dynamic ratio recommendation" tag.

Configuration Operations

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the name of the target real-time service to enter its details page. Switch to the **Intelligent O&M** tab to enter the configuration interface.
3. Turn on the monitoring switch and configure the monitoring algorithm parameters. The system will then automatically calculate and display the optimal P/D ratio recommendation based on real-time metrics and simulation algorithms.
4. Follow the prompt and click the optimize button to view the optimization suggestions provided by the system. Click the apply button to complete the intelligent O&M ratio tuning.

1.10 Logging and Monitoring

1.10.1 Viewing Real-Time Service Logs

Overview

ModelArts inference services allow you to see real-time logs and connect with LTS. Use these logs to debug model startups, check request inputs and outputs, find runtime errors, confirm service logic, and analyze performance issues.

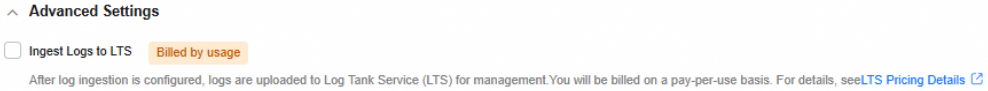
Table 1-51 Log description

Dimension	Real-Time Service Logs	LTS Integration
Log coverage	Displays only container standard output and error logs. It does not include platform scheduling logs.	Simultaneously collects container standard output, error logs, and Kubernetes events (pod events). It does not include platform scheduling logs.
Storage and historical capabilities	No persistence. Retains only the latest snippets of the currently running instance. Logs are lost and cannot be retroactively retrieved once an instance restarts or is destroyed.	Persistent storage with custom retention periods. Caches operational logs within the last 7 days by default and supports historical queries.
Core functions	Basic viewing and simple keyword search.	Multi-dimensional retrieval, data analysis, log alerting, and log archiving.
Use cases	Model debugging and instant troubleshooting for individual inference issues.	Production O&M, fault tracing, compliance auditing, and performance statistics.
Enabling method	Enabled automatically by default.	Manually configured during the service information configuration phase when deploying the real-time service.
Billing	Real-time logging itself is free of charge.	LTS is billed on a pay-per-use basis. For details, see **Log Tank Service Pricing Details**.

Viewing Real-Time Logs

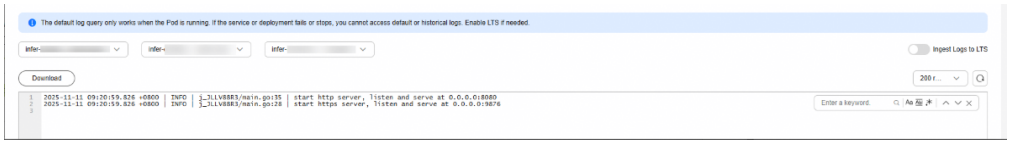
1. Create a real-time service by referring to **Deploying a Real-Time Inference Service Using a Single Node**. Do not select **Ingest Logs to LTS** in **Advanced Settings**.

   Figure 1-79 Ingest Logs to LTS not selected

2. After the service is in the **Running** state, click the service name to access its details page. Switch to the **Logs** tab and select the instance and Pod to view real-time service logs.

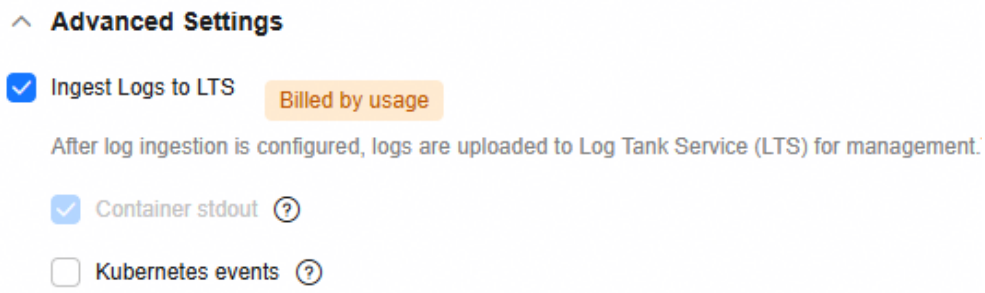
Figure 1-80 Viewing service logs



Viewing Logs Reported to LTS

1. When creating a resource pool, install the cloud native log collection plugin. For details, see **Creating a Dedicated Resource Pool**.
2. Create a real-time service by referring to **Configuring Deployment Settings**. Select **Ingest Logs to LTS** in **Advanced Settings**.

Figure 1-81 Ingest Logs to LTS



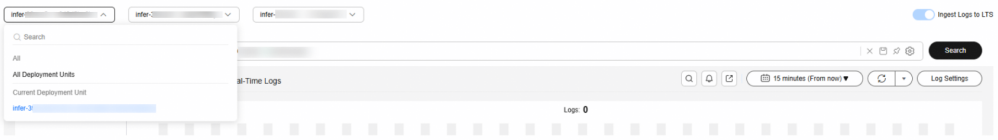
Alternatively, enable LTS when modifying a real-time service. To do so, locate the target service on the **Real-Time Services** page and choose **More** > **Modify** in the **Operation** column. Then, select **Ingest Logs to LTS** in **Advanced Settings**.

Parameter	Description
Container stdout	After you select **Ingest Logs to LTS**, **Container stdout** is selected by default and cannot be modified. Collect all container standard output and report it to Log Tank Service (LTS). The log retention period follows the setting of the corresponding log group, which is 30 days by default: • For dedicated resource pools, the LTS log groups and log streams are those created and selected during the installation of the resource pool log collection plugin. • For public resource pools, the system automatically creates LTS log groups and log streams. The naming conventions are: Log group: Modelarts-Infer-Log-Group-{NUM}; log stream: Inf-Stdout-{serviceId}.

Parameter	Description
Kubernetes events	After you select **Ingest Logs to LTS**, you can manually select **Kubernetes events**. Once enabled, Kubernetes events (pod events) will be collected and reported to LTS. By default, logs are retained for 7 days. You can view pod events in the event list on the inference service details page. For details, see **Viewing Events of a Real-Time Service**. If this option is not selected, you can only view pod events of the last hour in the event list on the inference service details page.

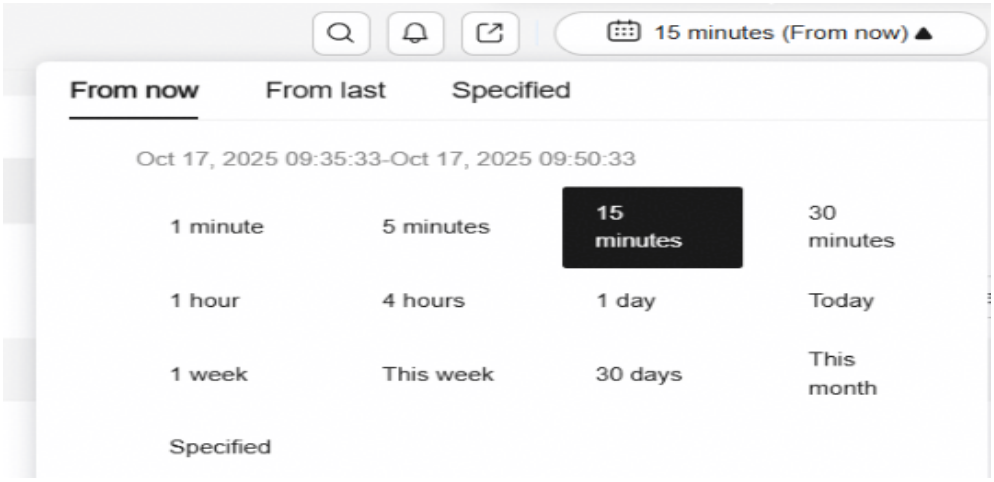
3. After the service is interconnected with LTS, access the service details page, and switch to the **Logs** tab to view logs.
4. Select the deployment, instance, and Pod to be viewed. Deleted ones can also be selected for fault locating.

Figure 1-82 Viewing instance logs

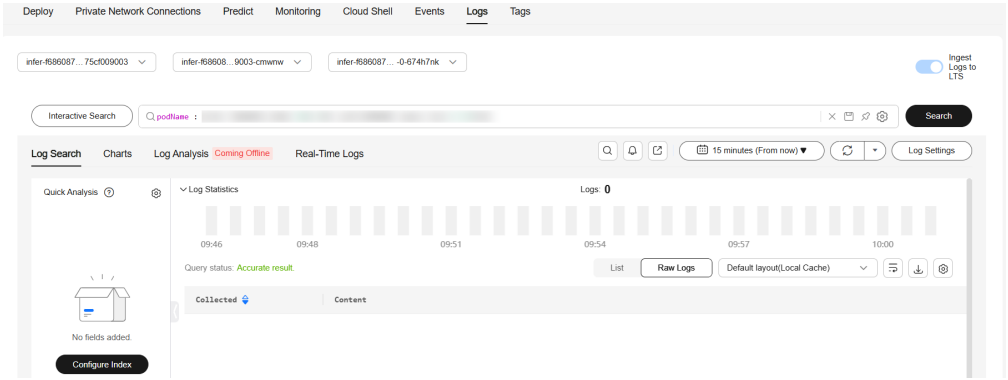


5. Select the time range of logs.

Figure 1-83 Selecting the time range of logs to be viewed



6. View log information.



- Log search: Search logs using specific keywords or phrases. Narrow your results by selecting a specific time range to find events and issues during that period.
- Statistical charts: After sending logs to LTS, use SQL analysis syntax to find important log data and view the results as statistical charts.
- Log analysis: Before searching for analyzing logs, set up structured data and indexing for them.
- Real-time logs: Once you connect your real-time service logs to LTS, they will be sent every minute. You can view these updates from the **Real-Time Logs** tab, where you can also easily search and analyze the data.

For details, see **Log Tank Service**.

7. To detect abnormal logs in a timely manner and further resolve issues, follow the steps in **Configuring LTS Log Alarms**.

Configuring LTS Log Alarms

1. After the service is interconnected with LTS, access the service details page, and switch to the **Logs** tab to view logs.
2. Click **Log Settings**. In the displayed dialog box, click the **Alarm Rules** tab.
3. Click **Create**. In the displayed dialog box, configure an alarm rule.

   For details about the parameters, see **Configuring Log Alarm Rules**.

   To add a notification when an alarm is generated, enable and create alarm notification rules on the LTS console. For details, see **Creating a Message Template on the LTS Console**.
4. Click **OK**.

   You can change the alarm settings in the log settings dialog box. For details, see **Follow-up Operations on Alarm Rules**.

FAQs

How Do I Promptly Detect Service Container Exceptions During Real-Time Service Deployment or Instance Restart?

During model service deployment, the system must download these files from the storage services and mount them to service containers to complete the deployment process. However, in actual operation, if any of the dependency storage services encounter issues, such as network interruptions, service unavailability, or insufficient disk space, the system may fail to detect these

exceptions effectively. This can lead to deployment failures or prolonged service unavailability, without any corresponding event reporting, thereby impacting service reliability and user experience.

To address this issue, the inference platform integrates with the LTS system. Through LTS log-based alerts, developers can be promptly notified of anomalies during the model weight file reading process, helping ensure smooth service deployment.

Rectify the fault by following *FAQs* > "How Do I Promptly Detect Service Container Exceptions During Real-Time Service Deployment or Instance Restart?"

1.10.2 Viewing Events of a Real-Time Service

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 service deployment and running process and accurately locate faults when a task exception occurs.

The supported event types include:

- Service events: Record the operational status and operations at the service level.
- Pod events: Record the lifecycle events and exception conditions of the underlying containers.

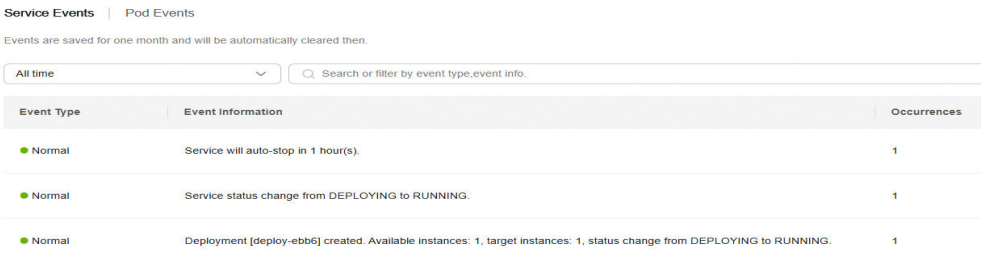
Procedure

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**. In the service list, click the target service name to go to the service details page.
2. View events in the **Events** tab.

   Events are classified into **service events** and **Pod Events**, which can be viewed by switching tabs.

   Service events record the status and operations at the service layer. Pod events record the lifecycle and exceptions of underlying containers.

   Figure 1-84 Viewing events



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

Service Events

Service events record key activities like starting, stopping, updating, or recovering services. These events help you monitor performance, understand past actions, solve issues, and improve settings. For details about common exception and warning events, see **Table 1-52** and **Table 1-53**. Normal events are not described here.

Table 1-52 Exceptional events

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Exception	Service deployment failed, %s.	Locate and rectify the fault based on the error information.
Exception	Service deployment timed out. Error: The deployment duration of the service exceeds *%s* minutes. The deployment failed.	Try again later or contact technical support.
Exception	Service deletion failed. Current status: %s, %s.	Locate and rectify the fault based on the error information.
Exception	Batch service deletion failed, %s.	Locate and rectify the fault based on the error information.
Exception	Service deletion timed out. Error message: Service deletion time exceeded %s minute. The deletion failed.	Try again later or contact technical support.
Exception	Service asynchronous task cancellation failed, %s.	Locate and rectify the fault based on the error information.
Exception	Creation of deployment [%s] failed, %s.	Try again later or contact technical support.
Exception	Update of deployment [%s] failed, %s.	Try again later or contact technical support.
Exception	The Volcano plugin for resource pool [%s] is unavailable.	Try again later or contact technical support.
Exception	Deletion of deployment [%s] failed, %s.	Locate and rectify the fault based on the error information.
Exception	Stopping of deployment [%s] failed, %s.	Locate and rectify the fault based on the error information.
Exception	Deletion of secret [%s] failed, %s.	Locate and rectify the fault based on the error information.
Exception	Verification of secret [%s] failed, %s.	Locate and rectify the fault based on the error information.

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Exception	Service configuration delivery failed, %s.	Locate and rectify the fault based on the error information.
Exception	Binding API key: %s failed, %s.	Locate and rectify the fault based on the error information.
Exception	Unbinding API key: %s failed, %s.	Locate and rectify the fault based on the error information.
Exception	Service startup failed, %s; status transitioned from %s to %s.	Service startup failures can happen for various reasons. For details about how to locate and rectify the fault, see **Failed to Start a Service**.
Exception	Service stop failed, %s.	Locate and rectify the fault based on the error information.
Exception	Service interruption failed. Current service status: %s, %s.	Locate and rectify the fault based on the error information.
Exception	Unable to release quota.	Try again later or contact technical support.
Exception	Interruption of deployment [%s] failed, %s; status transitioned from %s to %s.	Locate and rectify the fault based on the error information.
Exception	Interruption of deployment [%s] failed. Current deployment status: [%s], %s.	Locate and rectify the fault based on the error information.
Exception	Stopping of deployment [%s] failed, %s; status transitioned from %s to %s.	Locate and rectify the fault based on the error information.
Exception	Deletion of deployment [%s] failed, %s; status transitioned from %s to %s.	Locate and rectify the fault based on the error information.
Exception	Service update failed; the service will roll back, %s.	Locate and rectify the fault based on the error information.
Exception	Service update timed out. Error message: Service deployment time exceeded %s minutes; the service is rolled back.	Locate and rectify the fault based on the error information.
Exception	Service upgrade configuration failed, %s.	Locate and rectify the fault based on the error information.
Exception	Update of deployment [%s] failed; the deployment will roll back, %s.	Locate and rectify the fault based on the error information.

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Exception	The service fails to be rolled back. Current service status: *%s*, *%s*.	Locate and rectify the fault based on the error information.
Exception	Service [%s] rollback timed out. Error message: %s.	Locate and rectify the fault based on the error information.
Exception	Upgrade rollback of deployment [%s] failed. Current status: [%s].	Locate and rectify the fault based on the error information.
Exception	Upgrade rollback of deployment [%s] failed; status transitioned from %s to %s.	Locate and rectify the fault based on the error information.
Exception	Rollback of deployment [%s] timed out; version: [%s].	Locate and rectify the fault based on the error information.
Exception	Service status transitioned from %s to %s; anomalous deployment [%s]; error message: %s.	Locate and resolve the issue based on the error message. For more solutions, see **Abnormal Real-Time Service Deployment Instances**.
Exception	Status of deployment [%s] transitioned from %s to %s; error message: %s.	Locate and resolve the issue based on the error message. For more solutions, see **Abnormal Real-Time Service Deployment Instances**.
Exception	Service status transitioned from %s to %s.	Locate and resolve the issue based on the error message. For more solutions, see **Abnormal Real-Time Service Deployment Instances**.
Exception	Service detected a chip fault in resource pool [%s]: %s.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Service detected a switch fault in resource pool [%s]: %s.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Service detected a node fault in resource pool [%s]: %s.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Service detected a network service anomaly event: %s.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Exception	Failed to submit the task for deleting instance [%s]. Reason: %s.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Failed to delete instance [%s]. Try again later or contact service O&M personnel for a resolution.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Failed to delete instance [%s]. Reason: Retries reached the maximum number of attempts (%s).	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Failed to delete instance [%s]. Only 1 instance remaining currently.	Locate and resolve the issue based on the error message. If it cannot be resolved, contact technical support.
Exception	Creation of deployment [%s] failed, %s; status transitioned from %s to %s.	Try again later or contact technical support.
Exception	Startup of deployment [%s] failed, %s; status transitioned from %s to %s.	Locate and resolve the issue based on the error message. For more solutions, see **Failed to Start a Service**.
Exception	Manual scaling of deployment [%s] failed. Error message: %s.	Check the error information. If resources are insufficient, release other resources or reduce the scaling quantity.
Exception	An anomaly exists in the service components that deployment [%s] depends on. Rescheduling is in progress; the current service status may be inaccurate.	Try again later or contact technical support.

Table 1-53 Warning events

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Warning	Deployment [%s] has no [**Health Check** > **Startup Probe**] configured, which will prevent the model loading process from being monitored. The service may display a **Running** status but fail to handle predictions because the model is still loading.	Configure a health check startup probe for the corresponding deployment. For details, see **Real-Time Service Health Check**.
Warning	Checking the health status of some pods failed during the deployment *[%s]*. (Check the pod events for the cause.)	• During service deployment or modification, it is common that the probe check fails. If the fault persists, the image service cannot be started. Locate the fault based on the logs. • During service runtime, if a chip or network fault occurs, the readiness probe and liveness probe check may fail. If the number of retries exceeds the threshold, the pod will be restarted.
Warning	Failed to pull the pod image in deployment [%s]. (Check the pod events for the cause.)	• The specified image may not exist or has been deleted. Check whether the selected image exists. • If you create a service using an API, check whether the entered image path is correct and whether the current account has the permission to access the API.

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Warning	Failed to start the pod container in deployment [%s]. (Check the pod events for the cause.)	• Check whether the image architecture matches the resource pool architecture. For example, the resource pool of the Arm architecture requires the Arm image. • Check whether the configured resource specifications are proper and whether the resources are sufficient for starting the image. If model mounting is set, for example, in the mounting weight scenario, the memory must be greater than the total size of the mounted file. • Check whether the startup command configuration is correct. • Check whether the image can be started and work properly.

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
Warning	Pod scheduling failed in deployment [%s]. (Check the pod events for the cause.)	A scheduling failure event indicates that the scheduler fails to schedule the pod this time, but does not indicate that the pod cannot be scheduled. The scheduler continuously attempts to schedule the pod until the service deployment time expires. • Check whether the node resources are abundant. If "insufficient cpu/memory" is displayed in the pod event, the existing node resources cannot meet the requirements. • Check whether the node is tainted. If "taints" is displayed in the pod event, the node is tainted. • Check whether there are too many pods on the node. If "too many pods" is displayed in the pod event, the node is overloaded with pods. In this case, you need to stop some services. • Check the status of the model warmup task. When you deploy a real-time service and set the model source to **Pre-warmed model**, strong affinity rules are automatically configured to schedule pods only on pre-warmed nodes. The possible causes are as follows: – Insufficient nodes prevent scheduling the configured instances for model warmup. Add more nodes to fulfill the service requirements. – Pre-warmed nodes might lack enough resources for scheduling the current service. You can disable unnecessary services or scale out the resource pool. For details about how to stop a service deployment, see **Stopping a Service Deployment**. For details about how to scale out a resource pool, see **Resizing a Dedicated Resource Pool**. • Check the affinity scheduling configuration. If affinity scheduling is configured, the possible causes are as follows:

Event Type	Event Message (*xxx* and *%s* represent placeholders)	Solution
		– Affinity is configured for some nodes, and strong affinity is required. The available resources of the selected nodes cannot meet the pod requirements or the nodes are tainted. As a result, the pod cannot be scheduled. – Anti-affinity is configured for some nodes, and strong affinity is required. The available resources of the nodes other than the selected nodes cannot meet the pod requirements or the nodes are tainted. As a result, the pod cannot be scheduled.
Warning	Pod mounting failed during the deployment *[%s]*. Retrying... (Check the pod events for the cause.)	It takes some time to mount PVCs. If this error occurs in a short time, ignore it. If SFS Turbo shows a mounting failure, verify its association status on the console's **Network** page.
Warning	Model configuration loading failed for pods in service deployment unit [%s]. Retrying... Details: %s	Rectify the fault by scenario based on the detailed error information. For details, see **Table 1-54**.

Table 1-54 Mode configuration loading failure details and solutions

Error Details	Solution
Unknown error. Try again later. An unknown error has occurred. Please try again later	Try again later or contact technical support.
A system error (such as insufficient disk space or disk damage) occurs during the download of OBS files. A system-related error (such as disk full error, disk corruption, etc.) occurred during the OBS download process	Check whether disk alarms (such as disk pressure alarms) are generated on the nodes where the service resource pool is deployed. If yes, handle the alarms in a timely manner or expand the capacity.

Error Details	Solution
An exception (such as network or permission problems) occurs when downloading OBS files. A download error (such as network issues, permission issues, etc.) occurred during the OBS download process	This problem may be caused by network fluctuation. Try again later. The OBS permission and policy configuration may be incorrect. As a result, the OBS file fails to be downloaded when local mounting acceleration is enabled.
The disk space is insufficient before the OBS file is downloaded. Insufficient disk space detected before downloading the OBS file	When local storage acceleration is enabled, the system checks the disk space before downloading OBS files. If the disk space is insufficient for OBS files, this event is generated. You are advised to expand the disk capacity or clear the disk.
Model warmup file not found. The model warmup file does not exist	If the model warmup file is not found when the service is started during model warmup, the event information is displayed. Check the status of the model warmup task or create a warmup task again.
The model warmup task is not successful. The model warmup task status is not successful	Try again later or create a model warmup task again.
An exception occurred when mounting SFS Turbo. Check whether the configuration is correct. An error occurred while mounting the SFS Turbo. Please check whether the related configurations are correct	When local cache acceleration is enabled, an exception occurs during SFS Turbo mounting. On the **Network** page of the console, check whether the SFS Turbo file system is disassociated and whether its status is normal.
Mounting SFS Turbo failed because access to the mount directory timed out. Try again later. A timeout was detected while accessing the mount directory when mounting the SFS Turbo. Please try again later	An access error occurred in the directory during the SFS Turbo mounting process when local cache acceleration was enabled. On the **Network** page of the console, check whether the SFS Turbo file system is disassociated and whether the status is normal. If the SFS Turbo file system is not disassociated and the status is normal, restore the SFS Turbo file system and deploy the service again.

Pod Events

pod events monitor the lifecycles and errors of pods within a Kubernetes cluster. In Kubernetes, a pod serves as the smallest deployable unit. Every instance of a real-time service matches one pod. Choose a specific instance to see its related pod events. Pod events help you understand the status and exceptions of service instances. For details about common pod events, see **Table 1-55**.

If you turn on **Ingest Logs to LTS** and choose **Kubernetes events** while **setting up your service**, these events (pod events) will be sent to LTS. Logs are kept for the past seven days by default. You can see pod events from the last seven days.

If you do not choose this option, you can only see pod events from the last hour.

Table 1-55 Pod events

Type	Event	Description	Solution
Normal	SuccessfulCreate	The container is created.	N/A
Normal	Started	The container is started.	N/A
Normal	Scheduled	The pod is scheduled to the node.	N/A
Normal	SuccessfulMountVolume	The storage volume is mounted.	N/A
Normal	Pulling	The image is being pulled.	N/A
Normal	Pulled	The image is pulled.	N/A
Normal	Healthy	The container is in the healthy state.	N/A
Normal	Killing	The container is being terminated.	N/A

Type	Event	Description	Solution
Alarm	BackOffStart	The container fails to start.	• Check whether the image architecture matches the resource pool architecture. For example, the resource pool of the Arm architecture requires the Arm image. • Check whether the configured resource specifications are proper and whether the resources are sufficient for starting the image. If model mounting is set, for example, in the mounting weight scenario, the memory must be greater than the total size of the mounted file. • Check whether the startup command configuration is correct. • Check whether the image can be started and work properly.
Alarm	CrashLoopBackOff	The container is repeatedly restarted and then breaks down.	
Alarm	FailedPullImage	Pulling the image failed.	• The specified image may not exist or has been deleted. Check whether the selected image exists. • If you create a service using an API, check whether the entered image path is correct and whether the current account has the permission to access the API.
Alarm	BackOffPullImage	The image fails to be pulled again.	
Alarm	Unhealthy	The health check fails.	• During service deployment or modification, it is common that the probe check fails. If the fault persists, the image service cannot be started. Locate the fault based on the logs. • During service runtime, if a chip or network fault occurs, the readiness probe and liveness probe check may fail. If the number of retries exceeds the threshold, the pod will be restarted.

Type	Event	Description	Solution
Alarm	FailedScheduling	The pod cannot be scheduled to the node temporarily.	A scheduling failure event indicates that the scheduler fails to schedule the pod this time, but does not indicate that the pod cannot be scheduled. The scheduler continuously attempts to schedule the pod until the service deployment time expires. • Check whether the node resources are abundant. If "insufficient cpu/ memory" is displayed in the pod event, the existing node resources cannot meet the requirements. • Check whether the node is tainted. If "taints" is displayed in the pod event, the node is tainted. • Check the affinity scheduling configuration. If affinity scheduling is configured, the possible causes are as follows: – Affinity is configured for some nodes, and strong affinity is required. The available resources of the selected nodes cannot meet the pod requirements or the nodes are tainted. As a result, the pod cannot be scheduled. – Anti-affinity is configured for some nodes, and strong affinity is required. The available resources of the nodes other than the selected nodes cannot meet the pod requirements or the nodes are tainted. As a result, the pod cannot be scheduled.
Alarm	FailedMount	The storage volume fails to be mounted.	It takes some time to mount PVCs. If this error occurs in a short time, ignore it. If SFS Turbo shows a mounting failure, verify its association status on the console's **Network** page.

Type	Event	Description	Solution
Alarm	InfersInitContainerFailed (incident)	The event details are as follows: Infers init container checked failed, errcode: %s, errmsg: %s. Exception information reported during the pre-check or operation performed by the init container where inference is started.	Rectify the fault based on the handling suggestions in Table 1-54 for the error message.

1.10.3 Viewing Performance Metrics of a Real-Time Service on ModelArts

Overview

ModelArts inference monitoring provides a full-stack, end-to-end observability capability for real-time inference services. Covering four major dimensions (the resource layer, network layer, request layer, and model inference layer), it collects real-time operational data and presents it visually. It supports performance diagnosis, exception alerting, and capacity planning to ensure the stable and highly efficient operation of AI services.

Table 1-56 lists the supported monitoring dimensions and metrics.

Table 1-56 Supported monitoring dimensions

Dimension	Metric
Resource utilization monitoring	CPU/Memory: Real-time collection of CPU utilization, core count, memory utilization, and memory usage (MB) for inference instances to reflect basic resource load. GPU/NPU: Collects GPU/NPU utilization, VRAM utilization, and VRAM occupancy to precisely monitor AI acceleration hardware loads. This is optimized for LLMs and deep learning inference scenarios.

Dimension	Metric
Network traffic monitoring	Inbound/Outbound traffic rate: Real-time statistics of data received/sent rates (Byte/s) to identify network bandwidth bottlenecks and anomalous traffic anomalies. Connections: Real-time active connection counts and connection trends to evaluate concurrent handling capacity and detect long-connection leaks.
Request performance monitoring	Service request count: Aggregated total of 2xx (success), 4xx, and 5xx (exception) requests within a statistical period, providing a direct reflection of service availability. Service request QPS: Queries per second, measuring the service's concurrent processing capabilities. Service request latency: Average latency along with TP50/TP90/TP99 latency (ms) to pinpoint slow requests and performance bottlenecks. Principle: Intercepts incoming traffic at the inference service ingress, records request/response timestamps and status codes, and aggregates them per statistical period. Supports millisecond-level latency precision.
Specialized model inference monitoring (LLM adapted)	TTFT: The duration from when a request is initiated until the first token is returned. This is a core metric for LLM streaming inference that reflects model initialization and first-frame generation efficiency. TPOT: The generation latency for each subsequent token, used to measure the stability of continuous model inference. Token count statistics: Input, output, total token counts, and increments. This adapts to billing systems and helps analyze the scale of model inputs and outputs.

Viewing Different Types of Monitoring Metrics

Table 1-57 Methods for viewing different types of monitoring metrics

Viewing Method	Metrics	Use Case	Reference
ModelArts console	LLM inference service metrics (QPS, latency, TTFT, token counts, service status) and basic resource monitoring metrics.	Daily operations and service observation.	1. **Inference Metrics Viewable on the ModelArts Console** 2. **Viewing Metrics on the ModelArts Console** 3. **Viewing Metrics on a Custom Dashboard on the ModelArts Console**
AOM console	Underlying infrastructure resources (CPU/GPU/NPU/ nodes), full raw metrics, alarms, and custom dashboards.	O&M troubleshooting, capacity planning, and SLA assurance.	**Viewing Performance Metrics of a Real-Time Service on AOM**
Custom metrics	Internal model states, service-specific dimensions, performance breakdowns, cache/queues, and billing dimensions.	Deep LLM O&M scenarios that are not covered by default metrics.	**Custom Metric Collection**

Prerequisites

- Permission configuration: You must be configured with AOM read-only permissions. If you are using role/policy-based permissions, assign the AOM ReadOnlyAccess system policy. If you are using identity policy-based permissions, assign the AOMReadOnlyPolicy system identity policy.
- Service deployment: The real-time service must be deployed, with its status showing as **Running**, **Alarm**, or **Upgrading** (services in **Deploying** or **Stopped** status will not have complete monitoring data).

Constraints

The monitoring time span can be up to 15 days, and the statistical period can be 1 minute, 5 minutes, 15 minutes, or 1 hour.

Inference Metrics Viewable on the ModelArts Console

Table 1-58 Inference metrics viewable on the ModelArts console

Parameter	Description
Used CPU	Used CPU cores of a real-time service.
CPU Usage	CPU usage of a real-time service.
Used Memory	Used memory (unit: MB) of a real-time service.
Memory Usage	Memory usage of a real-time service.
GPU Usage	Available when GPU resources are used. GPU usage of a real-time service.
GPU Memory	Available when GPU resources are used. GPU memory utilization and usage of a real-time service.
NPU Usage	Available when NPU resources are used. NPU usage of a real-time service.
NPU Memory	Available when NPU resources are used. NPU memory utilization and usage of a real-time service.
Outbound Network Throughput	Inbound traffic rate of a real-time service, in Byte/s.
Inbound Network Throughput	Outbound traffic rate of a real-time service, in Byte/s.
Service Requests	Number of calls with various return codes (2xx, 4xx, 5xx) during a specific period for a real-time service. This represents the total call volume in that period. The 2xx code shows the number of successful calls.
Service Request QPS	QPS of a real-time service. The value is the total number of calls in a statistical period divided by the number of seconds in the selected period.
Connections	Real-time connection sampling data of a real-time service, which is used to provide the number of requests that are in the connection setup state.
Service Request Latency	Average request latency trend, TP50, TP90, and TP99 of a real-time service, in milliseconds (ms). TPxx represents the threshold value below which xx% of user requests fall for a given metric within the statistical interval. For example, TP90 = 300 ms indicates that 90% of service requests are completed within 300 milliseconds. The value is an estimated value, which may be a decimal.

Parameter	Description
Time to First Token	Time to first token in a period, including the average latency, TP50, TP90, and TP99. TPxx represents the threshold value below which xx% of user requests fall for a given metric within the statistical interval. For example, a TP90 value of 300 ms for time to first token indicates that 90% of user requests experience a time to first token within 300 milliseconds. The value is an estimated value, which may be a decimal.
Post-First-Token Latency	Time to each token in a period, including the average latency, TP50, TP90, and TP99. TPxx represents the threshold value below which xx% of user requests fall for a given metric within the statistical interval. For example, a TP90 value of 300 ms for non-first-token latency indicates that, within a single generation cycle, 90% of users experience per-token output latency within 300 milliseconds. The value is an estimated value, which may be a decimal.
Service Token Increment	Increment of tokens in a period, including the total number of input tokens, output tokens, and service tokens. Total number of service tokens = Total number of input tokens + Total number of output tokens
Service Input Tokens	TP50, TP90, and TP99 of the input tokens of real-time services in a period. TPxx represents the threshold value below which xx% of user requests fall for a given metric within the statistical interval. For example, if TP90 of the input tokens is 300, the input tokens of 90% of user requests are within 300. The value is an estimated value, which may be a decimal.
Output Tokens	TP50, TP90, and TP99 of the output tokens of real-time services in a period. TPxx represents the threshold value below which xx% of user requests fall for a given metric within the statistical interval. For example, if TP90 of the output tokens is 300, the output tokens of 90% of user requests are within 300. The value is an estimated value, which may be a decimal.

NOTICE

The CPU usage, memory usage, GPU usage and memory, NPU usage and memory are real-time data.

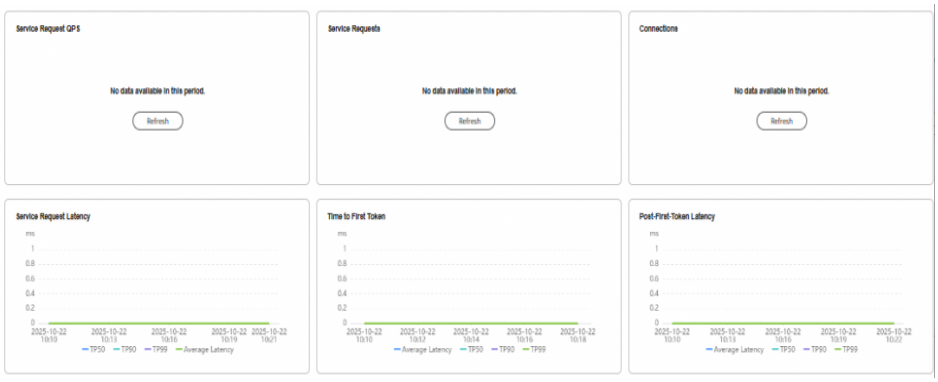
After creating the service, the system collects data on service requests, QPS, connections, request latency, token latency, and token count.

Monitoring metrics involving increments and TPxx are calculated incrementally. There may be too few sample points in a period and the calculated value may be **0**.

Viewing Metrics on the ModelArts Console

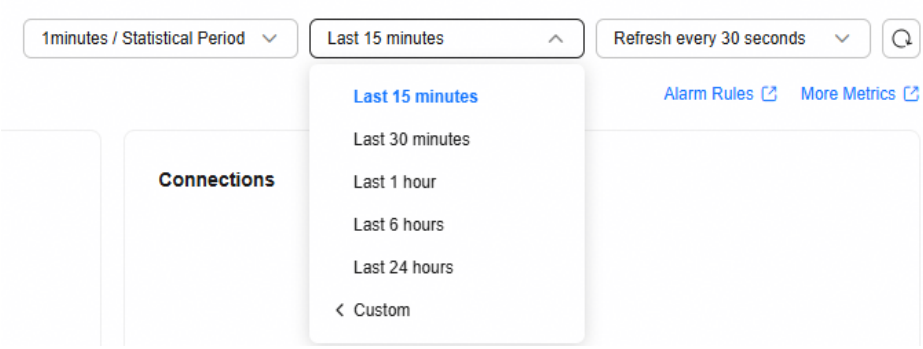
1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the name of the target real-time service to go to its details page.
3. Click the **Monitoring** tab. The system shows the last 15 minutes of service-level monitoring data by default, with statistics updated every minute.

Figure 1-85 Service-level monitoring information on the service details page



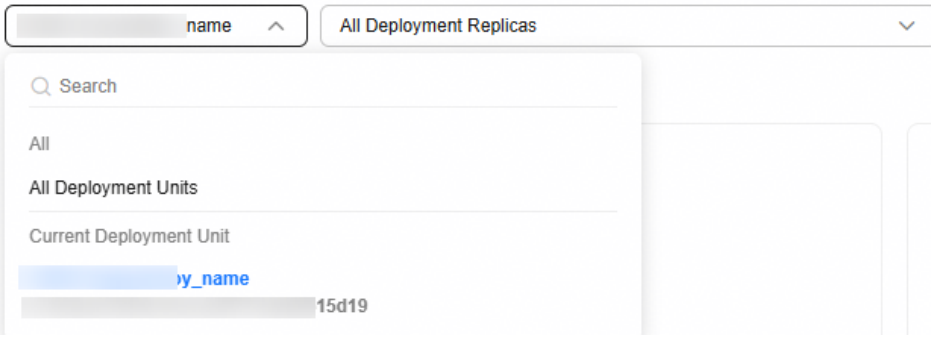
4. View monitoring data of the service and its versions in the **Monitoring** tab.
   - Choose a time range (up to 15 days) and a statistical period to view the needed monitoring data.

     Figure 1-86 Choosing a time range and a statistical period



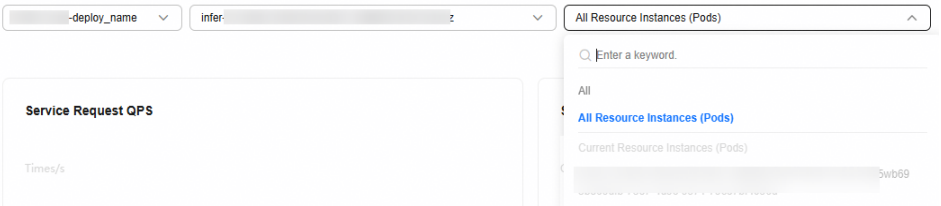
- Choose a deployment to view service deployment-level monitoring (only resource usage metrics are supported).

  Figure 1-87 Monitoring information of a service instance

- Choose a deployment, instance, and pod to view service pod-level monitoring (only resource usage metrics are supported).

  Figure 1-88 Monitoring information of service pods

Viewing Metrics on a Custom Dashboard on the ModelArts Console

To display metrics on the console in a personalized way, customize a dashboard based on your service needs.

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Inference** > **Real-Time Inference**.
2. Click the name of the target real-time service to go to its details page.
3. Click the **Monitoring** tab. Switch to the **Custom Dashboard** tab.

   The custom dashboard allows you to monitor metric values and trends in real time, and set alarm rules for important ones. For details about monitoring metrics and related operations, see **Observability Metric Browsing**.

1.10.4 Viewing Performance Metrics of a Real-Time Service on AOM

Overview

ModelArts sends all monitoring data for its inference services to AOM. The ModelArts console shows only a subset of this data. For refined O&M, alarms, and custom dashboards, you need to use AOM.

AOM is a one-stop, multi-dimensional O&M management platform for cloud applications. It monitors applications and cloud resources in real time, and can perform daily monitoring of the status of ModelArts real-time services and model workloads. You can obtain the metrics of each ModelArts real-time service and model on the AOM console. It takes a period of time to transmit and display monitored data. The statuses displayed on AOM are obtained 5 to 10 minutes before. You can view the monitoring data of a newly created real-time service 5 to 10 minutes later.

Service Access Metrics

The cloud service platform provides Application Operations Management (AOM) to help you better understand the statuses of ModelArts real-time services and model workloads. 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.

Table 1-59 Service access metrics

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Requests	Number of service requests	infer_service_request_total	Number of API calls of a real-time service in a collection period	Count	≥ 0	30s	Service API metric label
Real-time connections	Number of connections	infer_service_request_connect_count	Number of real-time connections of a real-time service at the time when the metric is collected	Count	≥ 0	30s	Service metric label
QPS	Service request QPS	infer_service_request_total	Real-time QPS of a real-time service at the time when the metric is collected	Queries/s	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Token	Number of tokens generated by a real-time service	infer_service_token_count	Number of tokens generated by a real-time service in a statistical period	Number	≥ 0	30s	Service metric label
	Average TTFT	infer_service_first_token_avg_cost	Average TTFT in a period	ms	≥ 0	30s	Service metric label
	Minimum TTFT	infer_service_first_token_min_cost	Minimum TTFT in a period	ms	≥ 0	30s	Service metric label
	Maximum TTFT	infer_service_first_token_max_cost	Maximum TTFT in a period	ms	≥ 0	30s	Service metric label
	Average output latency per token	infer_service_per_token_avg_cost	Average output latency of each token in a period (excluding the first token)	ms	≥ 0	30s	Service metric label
	Maximum output latency per token	infer_service_per_token_max_cost	Maximum output latency of each token in a period (excluding the first token)	ms	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
	Minimum output latency per token	infer_service_per_token_min_cost	Minimum output latency of each token in a period (excluding the first token)	ms	≥ 0	30s	Service metric label
	Average time consumed by each token	infer_service_token_latency_avg_cost	Average time consumed by each token in a period	ms	≥ 0	30s	Service metric label
	Minimum time consumed by each token	infer_service_token_latency_min_cost	Minimum time consumed by each token in a period	ms	≥ 0	30s	Service metric label
	Maximum time consumed by each token	infer_service_token_latency_max_cost	Maximum time consumed by each token in a period	ms	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Service latency	Request latency of a real-time service	infer_service_request_cost	This metric collects data on request latencies to determine TPxx and average latency. Average latency equals **infer_service_request_cost_sum** divided by **infer_service_request_cost_count**. To calculate TPxx latency, such as TP90, use this Prometheus statement: **histogram_quantile(0.90, avg(rate(infer_service_request_cost_bucket{service_id="6d26f238-52b0-4184-8b50-d621a0c80eb4"}[1m:30s]))by(le))**	ms	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
TTFT	TTFT	infer_service_first_token_cost	TTFT in a period, which can be used to calculate the average delay and TPxx. For the average latency calculation, see **sum(increase(infer_service_first_token_cost_sum{service_id="{service_id}"}[59999ms]))by(service_id)/sum(increase(infer_service_first_token_cost_count{service_id="{service_id}"}[59999ms]))by(service_id)**. For the TPxx latency calculation, see the Prometheus statement **histogram_quantile(XX%,avg(rate(infer_service_first_token_cost_bucket{service_id="{service_id}"}[1m:30s]))by(le))**.	ms	≥ 0	30s	Service metric dimension

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Post-first-token latency	Average latency of each output token	infer_service_per_token_cost	Latency of each output token in a period, which can be used to calculate the average delay and TPxx. For the average latency calculation, see **sum(increase(infer_service_per_token_cost_sum{service_id="{service_id}"}[59999ms]))by(service_id)/sum(increase(infer_service_per_token_cost_count{service_id="{service_id}"}[59999ms]))by(service_id)**. For the TPxx latency calculation, see the Prometheus statement **histogram_quantile(XX%,avg(rate(infer_service_per_token_cost_bucket{service_id="{service_id}"}[1m:30s]))by(le))**.	ms	≥ 0	30s	Service metric dimension

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Input tokens	Input tokens of real-time service requests	infer_service_input_token_quantity	Input tokens of requests, which can be used to calculate the average, TP50, TP90, and TP99 input tokens. For the average input token calculation, see **infer_service_input_token_quantity_sum/ infer_service_input_token_quantity_count**. For the TPxx input token calculation, see the Prometheus statement. For example, for TP90, the calculation method is **histogram_quantile(0.90,avg(rate(infer_service_input_token_quantity_bucket{service_id="6d26f238-52b0-4184-8b50-d621a0c80eb4"}[1m:30s]))by(le))**.	Number	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Output tokens	Output tokens of real-time service requests	infer_service_output_token_quantity	Output tokens of requests, which can be used to calculate the average, TP50, TP90, and TP99 output tokens. For the average output token calculation, see **infer_service_output_token_quantity_sum/ infer_service_output_token_quantity_count**. For the TPxx output token calculation, see the Prometheus statement. For example, for TP90, the calculation method is **histogram_quantile(0.90,avg(rate(infer_service_output_token_quantity_bucket{service_id="6d26f238-52b0-4184-8b50-d621a0c80eb4"}[1m:30s]))by(le))**.	Number	≥ 0	30s	Service metric label
Used cores	CPU cores used by a real-time service	ma_container_cpu_used_core	Real-time used CPU cores of a real-time service at the time when the metric is collected	Core	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
CPU utilization	CPU utilization of a real-time service	ma_container_cpu_util	Real-time CPU utilization of a real-time service at the time when the metric is collected	%	0%–100%	30s	Service metric label
Used memory	Memory usage of a real-time service	ma_container_memory_used_megabytes	Real-time memory usage of a real-time service at the time when the metric is collected	MB	≥ 0	30s	Service metric label
Memory utilization	Memory utilization of a real-time service	ma_container_memory_util	Real-time memory utilization of a real-time service at the time when the metric is collected	%	0%–100%	30s	Service metric label
NPU utilization	NPU utilization of a real-time service	ma_container_npu_util	Real-time NPU utilization of a real-time service at the time when the metric is collected	%	0%–100%	30s	Service metric label
Used NPU memory	NPU memory usage of a real-time service	ma_container_npu_memory_used_megabytes	Real-time NPU memory usage of a real-time service at the time when the metric is collected	MB	≥ 0	30s	Service metric label
NPU memory utilization	NPU memory utilization of a real-time service	ma_container_npu_memory_util	Real-time NPU memory utilization of a real-time service at the time when the metric is collected	%	0%–100%	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
GPU usage	GPU utilization of a real-time service	ma_container_gpu_util	Real-time GPU utilization of a real-time service at the time when the metric is collected	%	0%–100%	30s	Service metric label
Used GPU memory	GPU memory usage of a real-time service	ma_container_gpu_mem_used_megabytes	Real-time GPU memory usage of a real-time service at the time when the metric is collected	MB	≥ 0	30s	Service metric label
GPU memory utilization	GPU memory utilization of a real-time service	ma_container_gpu_mem_util	Real-time GPU memory utilization of a real-time service at the time when the metric is collected	%	0%–100%	30s	Service metric label
Inbound network throughput	Inbound network throughput of a real-time service	ma_container_network_receive_bytes	Real-time inbound network throughput of a real-time service at the time when the metric is collected	Byte/s	≥ 0	30s	Service metric label
Outbound network throughput	Inbound network throughput of a real-time service	ma_container_network_transmit_bytes	Real-time outbound network throughput of a real-time service at the time when the metric is collected	Byte/s	≥ 0	30s	Service metric label
Total TTFT	Total TTFT of a service in a period	infer_service_first_token_cost_sum	Total TTFT of a real-time service in a period	ms	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Total input tokens	Total service input tokens in a period	infer_service_input_token_quantity_sum	Total number of input tokens used by a real-time service in a period	ms	≥ 0	30s	Service metric label
Total output tokens	Total service output tokens in a period	infer_service_output_token_quantity_sum	Total number of output tokens used by a real-time service in a period	ms	≥ 0	30s	Service metric label
Total post-first-token latency	Total post-first-token latency of a service in a period	infer_service_per_token_cost_sum	Total non-first-token latency of a real-time service in a period	ms	≥ 0	30s	Service metric label
Total service request latency	Total request latency of a service in a period	infer_service_request_cost_sum	Total latency of real-time service requests in a period	ms	≥ 0	30s	Service metric label
TTFT trend	TTFT trend of a service	infer_service_first_token_cost_bucket	TTFT trend of a service	ms	≥ 0	30s	Service metric label
Service input tokens	Service input tokens	infer_service_input_token_quantity_bucket	Input tokens used by a service	Number	≥ 0	30s	Service metric label

Category	Name	Metric	Description	Unit	Value Range	Collection Period	Metric Label
Service output tokens	Service output tokens	infer_service_output_token_quantity_bucket	Output tokens used by a service	Number	≥ 0	30s	Service metric label
Post-first-token latency trend	Post-first-token latency trend	infer_service_per_token_cost_bucket	Post-first-token latency trend of a service	ms	≥ 0	30s	Service metric label
Service request latency trend	Service request latency trend	infer_service_request_cost_bucket	Request latency trend of a service	ms	≥ 0	30s	Service metric label

Label Metrics

Table 1-60 Service API metric labels

Label	Description
service_id	Inference service ID, for example, **9f322d5a-b1d2-4370-94df-5a87de27d36e**.
group_id	Inference service deployment ID.
project_id	Project ID of the account to which the user belongs.
code	Request return code, including 2xx, 4xx, and 5xx.
method	Inference request method.
path	Inference request path.
source	Pod ID for the request to pass through the dispatcher.
namespace	Metric name, which corresponds to the **Metric** column in **Table 1-59**.

Label	Description
instance_name	Inference service instance name.
service_name	Inference service name.

Table 1-61 Service metric labels

Label	Description
service_id	Inference service ID, for example, **9f322d5a-b1d2-4370-94df-5a87de27d36e**.
group_id	Inference service deployment ID.
project_id	Project ID of the account to which the user belongs.
source	Pod ID for the request to pass through the dispatcher.
namespace	Metric name, which corresponds to the **Metric** column in **Table 1-59**.

Viewing Metrics on AOM

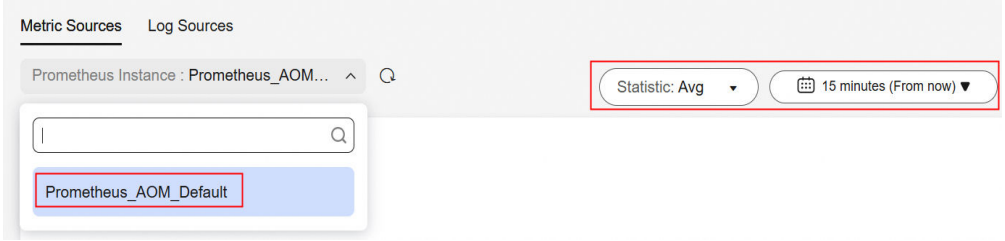
Prerequisites

- The ModelArts real-time service is running properly.
- 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.

Procedure

1. In the service list, choose **Management & Governance** > **Application Operations Management**.
2. In the navigation pane on the left, choose **Metric Browsing**. Set **Metric Sources** to **Prometheus_AOM_Default** and set the statistical method and statistical period as needed.

Figure 1-89 Selecting metric sources



3. At the bottom of the metric browsing page, select one or more target metrics by configuring **All metrics** or **Prometheus statement**.

   The following uses **All metrics** to obtain real-time service job metrics.

   - **Metric**: Enter a metric, for example, **ma_container_cpu_util**.
   - **Conditions**: Enter the condition (**instance_name**) and dimension value (real-time service job ID, which can be obtained from the real-time service details page on the ModelArts console). The system will show the metric monitoring curve for that job instantly.

   Figure 1-90 Setting real-time service job metrics

   For details about how to add metrics by configuring **Prometheus statement**, see **"Observability Metric Browsing" in Application Operations Management > User Guide (2.0)**.

Custom Metric Collection

Overview

If the default ModelArts metrics do not meet your needs, you can set up custom metrics (in Prometheus format) when deploying a real-time inference service. ModelArts will collect and send these metrics to AOM.

Constraints

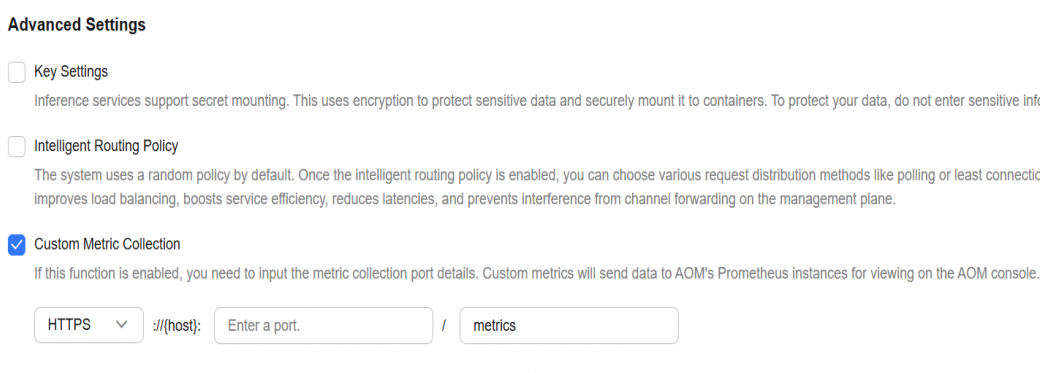
- ModelArts calls the HTTP API provided in the custom metric configuration every 10 seconds to obtain metric data.
- The metric data text returned by the HTTP API provided in the custom metric configuration cannot exceed 32 KB.

Configuration entry

Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**. When deploying a real-time service, check **Custom Metric Collection** under **Advanced Settings** during the deployment configuration phase. For details, see **Deploying a Real-Time Inference Service Using a Single Node**.

Select HTTPS (recommended) or HTTP as the protocol. Set the metric collection port to 8000 or 9090. Use **metrics** as the fixed API path.

Figure 1-91 Custom Metric Collection



Custom metrics are sent to the Prometheus instance in AOM. You can query these metrics in AOM. For details, see **Viewing Metrics on AOM**.

Data Format of Custom Metrics

The format of custom metrics data must comply with the open metrics specifications. That is, the format of each metric must be:

```
<metric_name>{<tag_name>=<tag_value>,...} <sample_value> [timestamp_in_millisecond]
```

The following shows an example (the comment starts with #, which is optional):

```
# HELP http_requests_total The total number of HTTP requests.
# TYPE http_requests_total gauge
html_http_requests_total{method="post",code="200"} 1656 1686660980680
html_http_requests_total{method="post",code="400"} 2 1686660980681
```

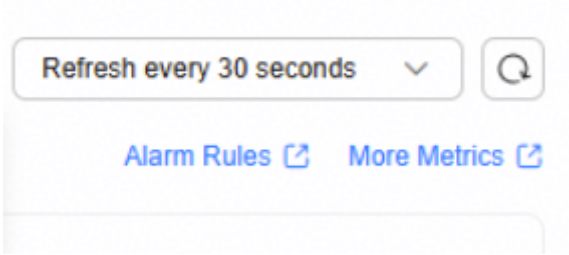
Configuring Alarms on AOM

Configuring alarms automatically detects issues like service resource problems and faulty requests, and sends timely alerts. This removes the need for constant human checks, accelerates issue resolution, ensures smooth service, and helps improve maintenance policies.

Alarm configuration path 1:

1. Log in to the **ModelArts console** and choose **Model Inference** > **Real-Time Inference**.
2. Click the name of the target real-time service in the service list. On the service details page, go to the **Monitoring** tab and click **Alarm Rules**. In the AOM console, configure alarm rules and notifications. For details, see **AOM Alarm Rule Overview**.

Figure 1-92 Alarm Rules



Alarm configuration path 2:

Log in to the **AOM console**. In the AOM console, configure alarm rules and notifications. For details, see **AOM Alarm Rule Overview**.

1.10.5 Using CTS to Audit Inference Service Operations

ModelArts can connect to CTS. With CTS, you can obtain operations associated with ModelArts for later query, audit, and backtrack operations.

Prerequisites

CTS has been enabled. For details, see **Cloud Trace Service User Guide**.

Key Inference Deployment (New Version) Operations Traced by CTS

Table 1-62 Key inference deployment (new version) operations traced by CTS

Operation	Resource Type	Trace
Creating an API key	apikeyV2	CreateApiKeyV2
Obtaining API keys	apikeyV2	ListApiKeyV2
Deleting an API key	apikeyV2	DeleteApiKeyV2
Binding an API key	apikeyV2	BindApiKeyV2
Unbinding an API key	apikeyV2	UnbindApiKeyV2
Batch binding API keys	apikeyV2	BindApiKeyV2
Batch unbinding API keys	apikeyV2	UnbindApiKeyV2
Creating a service deployment	serviceV2	CreateDeployment
Updating a service deployment	serviceV2	UpdateDeployment
Obtaining service deployment details	serviceV2	GetDeployment
Starting a service deployment	serviceV2	StartDeployment
Stopping a service deployment	serviceV2	StopDeployment
Deleting a service deployment	serviceV2	DeleteDeployment
Interrupting a service deployment	serviceV2	InterruptDeployment

Operation	Resource Type	Trace
Obtaining service deployments	serviceV2	ListDeployment
Scaling instances	serviceV2	ScaleDeployment
Obtaining service instances	serviceV2	ListServiceV2Instances
Obtaining service pods	serviceV2	ListServiceV2Pods
Deleting a service instance	serviceV2	DeleteServiceV2Instance
Deleting a pod	serviceV2	DeleteServiceV2Pod
Obtaining pod events	serviceV2	ListServiceV2PodEvents
Switching deployment versions	serviceV2	SwitchDeploymentVersion
Obtaining details about a real-time service deployment version	serviceV2	GetDeploymentVersion
Obtaining deployment versions	serviceV2	ListDeploymentVersion
Deleting a real-time service deployment version	serviceV2	DeleteDeploymentVersion
Creating an HPA policy	serviceV2	CreateHpa
Viewing an HPA policy	serviceV2	GetHpaV2
Deleting an HPA policy	serviceV2	DeleteHpa
Modifying an HPA policy	serviceV2	UpdateHpa
Obtaining HPA events	serviceV2	ListHpaEventV2
Querying details about a managed cluster	Cluster	GetClusterV2
Creating a service	serviceV2	CreateServiceV2
Starting a service	serviceV2	StartServiceV2
Updating a service	serviceV2	UpdateServiceV2
Obtaining service details	serviceV2	GetServiceV2
Stopping a service	serviceV2	StopServiceV2
Interrupting a service	serviceV2	InterruptServiceV2
Deleting a service	serviceV2	DeleteServiceV2

Operation	Resource Type	Trace
Obtaining services	serviceV2	ListServiceV2
Obtaining events of a specified service	serviceV2	ListEventsV2
Obtaining available flavors	ServiceFlavor	ListServiceFlavor
Batch creating resource tags	ServiceTag	CreateServiceTag
Batch deleting resource tags	ServiceTag	DeleteServiceTag
Obtaining resource tags	ServiceTag	ListResourceServiceTag
Obtaining tags of a project	ServiceTag	ListProjectServiceTag
Obtaining resource instances	ServiceTag	ListServiceTagResources
Obtaining the number of resource instances	ServiceTag	CountServiceTagInstance
Switching real-time service versions	serviceV2	SwitchServiceV2Version
Obtaining details about real-time service versions	serviceV2	GetServiceV2Version
Obtaining real-time service versions	serviceV2	ListServiceV2Version
Deleting a real-time service version	serviceV2	DeleteServiceV2Version
Creating a private network request	intranetConnection	CreateIntranetConnection
Operating a private network request	intranetConnection	OperateIntranetConnection
Obtaining private network access requests	intranetConnection	ListApprovalRequests
Obtaining private network access approvals	intranetConnection	ListApprovalReviews
Deleting a private network connection	intranetConnection	DeleteIntranetConnection
Modifying a request for adding a custom private network URL	intranetConnection	OperateIntranetConnection

Operation	Resource Type	Trace
Querying quotas	ProjectQuota	GetProjectQuota

For details about how to view audit logs in CTS, see **Querying Traces in CTS**.

Viewing Traces

1. Log in to the **CTS console**.
2. Click  in the upper left corner and select the target region.
3. In the navigation pane on the left, choose **Trace List**.

   Each time you log in to the CTS console, the new edition is displayed by default. Click **Old Edition** in the upper right corner to switch to the trace list of the old edition.
4. Filter traces to view information about the target traces.

   For details, see **Querying Traces in CTS**.

   You can call the API to get traces in the trace list to view resource operation records. For details, see **Querying a Trace List**.

2 Inference Deployment (Old Version)

2.1 Overview

Inference deployment is an important part of AI and machine learning. Inference deployment moves a trained machine learning or deep learning model from the development environment to the production environment. This allows the model to predict outcomes for new, unseen data. The goal is to ensure the model works efficiently and reliably in real-world applications, meeting real-time and resource needs. Simply put, it makes the model operational, providing outputs like image classification, text sentiment analysis, and trend predictions based on input data.

ModelArts offers you compute, storage, and network resources to deploy, use, manage, and monitor your AI model after development. ModelArts supports cloud, edge, and on-device deployments. It also supports real-time, batch, and edge inference.

Deployment Modes

ModelArts supports cloud deployments. It also supports real-time and batch inference.

Cloud-based deployment means deploying and running inference services on cloud servers. This works well for tasks needing lots of compute and data.

- Real-time inference: Processes a single request instantly and returns the result right away. ModelArts allows you to deploy a model as a web service that provides a real-time test UI and monitoring capabilities. This service provides a callable API. Real-time inference is used in situations that need fast responses, like online intelligent customer service and autonomous driving decisions.
- Batch inference: Processes multiple inputs at once and returns all results together. ModelArts allows you to deploy a model as a batch service. This service runs inference on batch data and stops automatically when done. Batch inference works well for offline tasks like big data analysis, batch data labeling, and model evaluation.

Description

Deploying a service in ModelArts uses compute and storage resources, which are billed. Compute resources are billed for running the inference service. Storage resources are billed for storing data in OBS. For details, see **Table 2-1**.

Table 2-1 Billing items

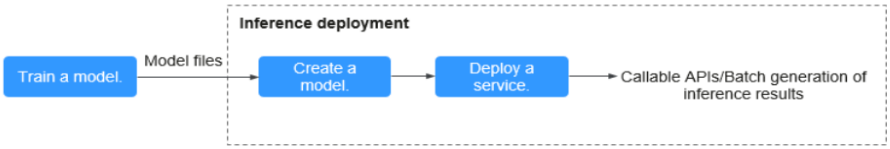
Billing Item		Description	Billing Mode	Billing Formula
Compute resource	Public resource pools	Usage of compute resources. For details, see **ModelArts Pricing Details**.	Pay-per-use	Specification unit price x Number of compute nodes x Usage duration
	Dedicated resource pools	Fees for dedicated resource pools are paid upfront upon purchase. There are no additional charges for service deployment. For details about dedicated resource pool fees, see **Dedicated Resource Pool Billing Items**.	N/A	N/A
Storage resource	Object Storage Service (OBS)	OBS is used to store the input and output data for service deployment. For details, see **Object Storage Service Pricing Details**. **CAUTION** OBS resources for storing data are continuously billed. To stop billing, delete the data stored in OBS.	Pay-per-use Yearly/ Monthly	Creating an OBS bucket is free of charge. You pay only for the storage capacity and duration you actually use.
Event notification (billed only when enabled)		This function uses Simple Message Notification (SMN) to send a message to you when the event you selected occurs. To use this function, enable event notification when creating a training job. For pricing details, see **Simple Message Notification Pricing Details**.	Pay by actual usage	• SMS: SMS notifications • Email: Email notifications + Downstream Internet traffic • HTTP or HTTPS: HTTP or HTTPS notifications + Downstream Internet traffic

Billing Item	Description	Billing Mode	Billing Formula
Run logs (billed only when enabled)	**Log Tank Service (LTS)** collects, analyzes, and stores logs. If **Runtime Log Output** is enabled during service deployment, you will be billed if the log data exceeds the LTS free quota. For details, see **Log Tank Service Pricing Details**.	Pay by actual log size	After the free quota is exceeded, you are billed based on the actual log volume and retention duration.

Inference Deployment Process

You can import and deploy AI models as inference services. These services can be integrated into your IT platform by calling APIs or generate batch results.

Figure 2-1 Introduction to inference

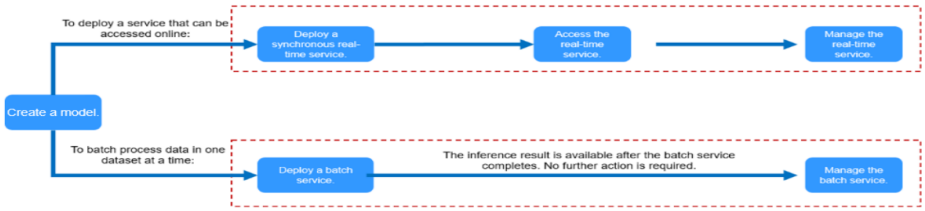


1. Prepare inference resources: Select required resources based on the site requirements. ModelArts provides you with public resource pools and dedicated resource pools. To use dedicated compute resources, you need to purchase and create a dedicated resource pool first. For details, see **Creating a Dedicated Resource Pool**.
2. Train a model: Models can be trained in ModelArts or your local development environment. A locally developed model must be uploaded to Huawei Cloud OBS.
3. Create a model: Import the model file and inference file to the ModelArts model repository and manage them by version. Use these files to build an executable model.
4. Deploy a service: Deploy the model as a service type based on your needs.
   - **Deploying a Model as Real-Time Inference Jobs**

     Deploy a model as a web service with real-time UI and monitoring. This service provides you a callable API.
   - **Deploying a Model as a Batch Inference Service**

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

Figure 2-2 Different inference scenarios



2.2 Creating a Model

2.2.1 Creation Methods

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.

With ModelArts, you can create models using meta models from training jobs, OBS, or container images, and centrally manage all iterated and debugged models.

Constraints

- After deploying a model in an ExeML project, it is automatically added to the model list. ExeML-generated models can only be deployed, not downloaded.
- All users can create models and manage model versions at no cost.

Meta Model Sources

- **Importing a Meta Model from a Training Job**: Create a training job in ModelArts to train a model. After obtaining a desired model, use it to create a model for service deployment.
- **Importing a Meta Model 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 for service deployment.
- **Importing a Meta Model 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, and use the image to create a model for service deployment.

Supported AI Engines for ModelArts Inference

If you import a model from OBS to ModelArts, the following AI engines and versions are supported.

NOTE

- A runtime environment of a unified image is named in the following format: *<AI engine and version> - <Hardware and version: CPU, CUDA, or CANN> - <Python version> - <OS version> - <CPU architecture>*
- Each preset AI engine has its default model start command. Do not modify it unless necessary.

Table 2-2 Supported AI engines, their runtime environments, and default start commands

Engine	Runtime Environment	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**. • The model can run on both CPUs and GPUs when using **python3.6**, **python2.7**, or **tf2.1-python3.7**. If the runtime environment has a suffix of **cpu** or **gpu**, the model can only run on CPUs or GPUs respectively. • The default runtime environment is **python2.7**. • Default start command: **sh /home/mind/run.sh**
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 environment is **python2.7**. • **python2.7** and **python3.6** can only be used to run models on CPUs. • Default start command: **bash /home/work/predict/bin/run.sh**
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 environment is **python2.7**. • **python2.7** and **python3.6** can only be used to run models on CPUs. • Default start command: **bash /home/work/predict/bin/run.sh**

Engine	Runtime Environment	Note
XGBoost	python2.7 (unavailable soon) python3.6 (unavailable soon)	• XGBoost 0.80 is used in **python2.7** and **python3.6**. • The default runtime environment is **python2.7**. • **python2.7** and **python3.6** can only be used to run models on CPUs. • Default start command: **bash /home/work/predict/bin/run.sh**
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**. • The model can run on both CPUs and GPUs when using **python2.7**, **python3.6**, **python3.7**, **pytorch1.4-python3.7**, or **pytorch1.5-python3.7**. • The default runtime environment is **python2.7**. • Default start command: **sh /home/mind/run.sh**
MindSpore	aarch64 (recommended)	**aarch64** can only be used to run models on Snt3 chips. • Default start command: **sh /home/mind/run.sh**

2.2.2 Importing a Meta Model from a Training Job

Create a training job in ModelArts to obtain a satisfactory model. The model can then be imported to create an AI application for centralized management. The application can be quickly deployed as a service.

Constraints

- You can directly import a model generated from a training job that uses a subscribed algorithm to ModelArts, without needing 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 a Model**.

Prerequisites

- The training job has been executed, and the model has been stored in the OBS directory where the training output is stored (the input parameter is **train_url**).
- If the training job uses a mainstream framework or custom image, upload the inference code and configuration file to the model storage directory by referring to **Model Package Structure**.
- The OBS directory you use must be in the same region as ModelArts.

Procedure

1. Log in to the ModelArts console and choose **Model Management** in the navigation pane on the left.
2. Click **Create Model**.
3. Configure parameters.
   a. Set basic information about the model. For details about the parameters, see **Table 2-3**.

Table 2-3 Basic information

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

   b. Set **Meta Model Source** to **Training job**. For details, see **Table 2-4**.

Figure 2-3 Importing a meta model from a training job

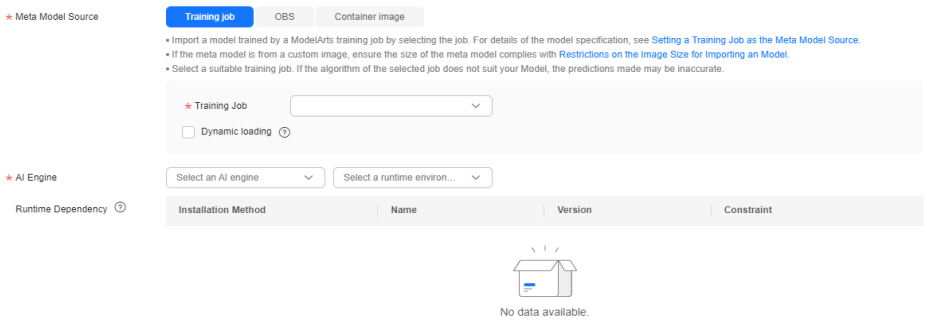


Table 2-4 Meta model source parameters

Parameter	Description
Meta Model Source	Select **Training job**. • Choose a training job from the **Training Job** drop-down list. • **Dynamic loading**: You can enable it for quick model deployment and update. When it is enabled, model files and runtime dependencies are only pulled during an actual deployment. Enable this feature if a single model file is larger than 5 GB.
AI Engine	Inference engine used by the meta model, which is automatically matched based on the training job you select.
Inference Code	Inference code customizing the inference logic of the model. You can directly copy the inference code URL for use.
Runtime Dependency	Dependencies that the selected model has on the environment. For example, if you need to install **tensorflow** using **pip**, make sure the version is 1.8.0 or newer.
Model Description	Model descriptions to help other developers better understand and use your model. Click **Add Model Description** and set the document name and URL. You can add up to three model descriptions.
Deployment Type	Choose the service types for model deployment. The service types you select will be the only options available for deployment. For example, selecting **Real-Time Services** means the model can only be deployed as real-time services.

c. Confirm the configurations and click **Create now**.

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

Follow-Up Operations

Deploying a service: In the model list, click **Deploy** in the **Operation** column of the target model. Locate the target version, click **Deploy** and choose a service type selected during model creation.

2.2.3 Importing a Meta Model from OBS

Import a model trained using a mainstream AI engine to ModelArts to create a model for centralized management.

Constraints

- The imported model, inference code, and configuration file must comply with ModelArts model package specifications. For details, see **Model Package Structure**, **Specifications for Editing a Model Configuration File**, and **Specifications for Writing a Model Inference Code 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 a Model**.
- If you use custom policies as an IAM user and do not use OBS Administrator policy, add these actions:
  - PutObject (uploading using PUT and POST, uploading parts, and initializing and merging uploaded parts)
  - GetObject (obtaining object content and metadata)
  - GetObjectVersion (obtaining the content and metadata of a specified object version)
  - GetObjectAcl (obtaining object ACL)
  - PutObjectAcl (configuring the object ACL)
  - ListBucket (listing objects in the bucket and obtaining the bucket metadata)

Prerequisites

- The trained model uses an AI engine supported by ModelArts. For details, see **Supported AI Engines for ModelArts Inference**.
- The model package, inference code, and configuration file have been uploaded to OBS.
- The OBS directory you use and ModelArts are in the same region.

Procedure

1. Log in to the **ModelArts console**. In the navigation pane on the left, choose **Model Management**.
2. Click **Create Model**.
3. Configure parameters.
   a. Set basic information about the model. For details about the parameters, see **Table 2-5**.

Table 2-5 Basic information

Parameter	Description
Name	Model name. The value can contain 1 to 64 visible characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.

Parameter	Description
Version	Model version. The default value is **0.0.1** for the first import. **NOTE** After a model is created, you can **create new versions** using different meta models for optimization.
Description	Brief description of the model.

b. Set **Meta Model Source** to **OBS**. For details, see **Table 2-6**.

To import a meta model from OBS, edit the inference code and configuration file by following **model package specifications** and place the inference code and configuration file in the **model** folder storing the meta model. If the selected directory does not comply with the model package specifications, the model cannot be created.

NOTE

Encrypt sensitive data before saving it to your OBS bucket.

Figure 2-4 Importing a meta model from OBS

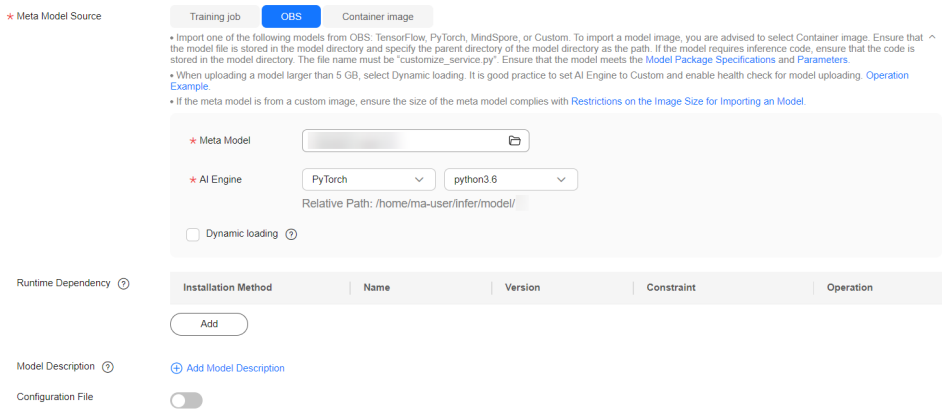


Table 2-6 Meta model source parameters

Parameter	Description
Meta Model Source	Select **OBS**.
Meta Model	OBS path for storing the meta model. The OBS path cannot contain spaces. Otherwise, the model creation will fail.
AI Engine	AI engine, which is automatically set according to the model storage path you select, used by the meta model.

Parameter	Description
Container API	This parameter is displayed when **AI Engine** is set to **Custom**. Set the protocol and port number of the inference API defined by the model. The default values are **HTTPS** and **8080**, respectively.

Parameter	Description
Health Check	Specifies health check on a model. This parameter is displayed when **AI Engine** is set to **Custom**. Once you select a non-custom engine and runtime environment, this parameter is displayed if this engine supports health check. Once you select a custom engine, you must select a container image for the engine package. The health check can be set up only if the container image includes a health check API. Otherwise, the model creation will fail. The following probes are supported: ● **Startup Probe**: This probe checks if the application instance has started. If a startup probe is provided, all other probes are disabled until it succeeds. If the startup probe fails, the instance is restarted. If no startup probe is provided, the default status is **Success**. ● **Readiness Probe**: This probe verifies whether the application instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. ● **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. The parameters of the three types of probes are as follows: ● **Check Mode**: Retain the default setting **HTTP request**. ● **Health Check URL**: Retain the default setting **/health**. ● **Health Check Period (s)**: Enter an integer ranging from 1 to 2147483647. ● **Delay (s)**: Set a delay for the health check to occur after the instance has started. The value should be an integer between 0 and 2147483647. ● **Timeout (s)**: Set the timeout interval for each health check. The value should be an integer between 0 and 2147483647. ● **Maximum Failures**: Enter an integer ranging from 1 to 2147483647. If the service fails the specified number of consecutive health checks during startup, it will enter the abnormal state. If the service fails the specified number of consecutive health checks during operation, it will enter the alarm state.

Parameter	Description
	NOTE To use a custom engine to create a model, 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 enabled for a model, the associated services will stop three minutes after receiving the stop instruction.
Dynamic loading	You can enable it for quick model deployment and update. When it is enabled, model files and runtime dependencies are only pulled during an actual deployment. Enable this feature if a single model file is larger than 5 GB.
Runtime Dependency	Dependencies that the selected model has on the environment. For example, if you need to install **tensorflow** using **pip**, make sure the version is 1.8.0 or newer.
Model Description	Model descriptions to help other developers better understand and use your model. Click **Add Model Description** and set the document name and URL. You can add up to three descriptions.
Configuration File	The system associates the configuration file stored in OBS by default. After enabling this feature, you can review and edit the model configuration file. **NOTE** This feature is to be discontinued. After that, you can modify the model configuration by setting **AI Engine**, **Runtime Dependency**, and **API Configuration**.
Deployment Type	Choose the service types for model deployment. The service types you select will be the only options available for deployment. For example, selecting **Real-Time Services** means the model can only be deployed as real-time services.
Start Command	Customizable start command of the model. This parameter is optional. When using a preset AI engine, the default start command is used if no start command is entered. For details, see **Table 2-2**. If a start command is entered, it replaces the default command. **NOTE** Start commands containing $, |, >, <, `, !, \n, \, ?, -v, --volume, --mount, --tmpfs, --privileged, or --cap-add will be emptied when a model is being published.

Parameter	Description
API Configuration	After enabling this feature, you can edit RESTful APIs to define the input and output formats of a model. The model APIs must comply with ModelArts specifications. For details, see the **apis** parameter description in **Specifications for Editing a Model Configuration File**. For details about the code example, see **Code Example of apis Parameters**.

c. Confirm the configurations and click **Create now**.

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

Follow-Up Operations

Deploying a service: In the model list, click **Deploy** in the **Operation** column of the target model. Locate the target version, click **Deploy** and choose a service type selected during model 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 from custom images.

Constraints

- For details about the specifications and description of custom images, see **Specifications for Custom Images Used for Importing Models**.
- 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 a model**.
- You must update the custom image version or enable image replication when creating a model. Otherwise, new models will still use the old image version.

Prerequisites

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

Procedure

1. Log in to the **ModelArts console**. In the navigation pane on the left, choose **Model Management**.
2. Click **Create Model**.
3. Configure parameters.
   a. Set basic information about the model. For details about the parameters, see **Table 2-7**.

Table 2-7 Basic information

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

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

Figure 2-5 Importing a meta model from a container image

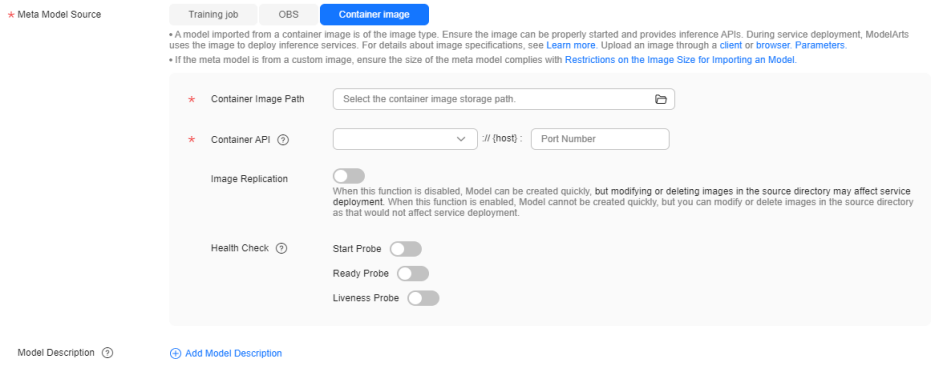


Table 2-8 Meta model source parameters

Parameter	Description
Container Image Path	Click  to import the container image. You do not need to use **swr_location** in the configuration file to specify the image location. For details about how to create a custom image, see **Specifications for Custom Images Used for Importing Models**. **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 by other accounts are not supported). ModelArts will deploy the image as an inference service. Ensure that your image can be properly started and provide an inference API.
Container API	Set the protocol and port number of the inference API defined by the model.

Parameter	Description
Image Replication	Indicates whether to copy the model image in the container image to ModelArts. ● After this feature is disabled, the model image is not copied, models can be rapidly created, but modifying or deleting an image in the SWR source directory will affect service deployment. ● After this feature is enabled, the model image is copied, models cannot be rapidly created, and modifying or deleting an image in the SWR source directory will not affect service deployment. **NOTE** You must enable this feature if you want to use images shared by others. Otherwise, models will fail to be created.

Parameter	Description
Health Check	Specifies health check on a model. This parameter is configurable only when a health check API is configured in the custom image. Otherwise, the model creation will fail. The following probes are supported: • **Startup Probe**: This probe checks if the application instance has started. If a startup probe is provided, all other probes are disabled until it succeeds. If the startup probe fails, the instance is restarted. If no startup probe is provided, the default status is **Success**. • **Readiness Probe**: This probe verifies whether the application instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. The parameters of the three types of probes are as follows: • **Check Mode**: Select **HTTP request** or **Command**. • **Health Check URL**: Enter the health check URL, which defaults to **/health**. This parameter is displayed when **Check Mode** is set to **HTTP request**. • **Health Check Command**: Enter the health check command. This parameter is displayed when **Check Mode** is set to **Command**. • **Health Check Period (s)**: Enter an integer ranging from 1 to 2147483647. • **Delay (s)**: Set a delay for the health check to occur after the instance has started. The value should be an integer between 0 and 2147483647. • **Timeout (s)**: Set the timeout interval for each health check. The value should be an integer between 0 and 2147483647. • **Maximum Failures**: Enter an integer ranging from 1 to 2147483647. If the service fails the specified number of consecutive health checks during startup, it will enter the abnormal state. If the service fails the specified number of

Parameter	Description
	consecutive health checks during operation, it will enter the alarm state. **NOTE** If health check is enabled for a model, the associated services will stop three minutes after receiving the stop instruction.
Model Description	Model descriptions to help other developers better understand and use your model. Click **Add Model Description** and set the document name and URL. You can add up to three descriptions.
Deployment Type	Choose the service types for model deployment. The service types you select will be the only options available for deployment. For example, selecting **Real-Time Services** means the model can only be deployed as real-time services.
Start Command	Customizable start command of a model. The path set in the boot command must be the same as the actual image path. **NOTE** Start commands containing $, |, >, <, `, !, \n, \, ?, -v, --volume, --mount, --tmpfs, --privileged, or --cap-add will be emptied when a model is being published.
API Configuration	After enabling this feature, you can edit RESTful APIs to define the input and output formats of a model. The model APIs must comply with ModelArts specifications. For details, see the **apis** parameter description in **Specifications for Editing a Model Configuration File**. For details about the code example, see **Code Example of apis Parameters**.

c. Confirm the configurations and click **Create now**.

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

Follow-Up Operations

Deploying a service: In the model list, click **Deploy** in the **Operation** column of the target model. Locate the target version, click **Deploy** and choose a service type selected during model creation.

2.3 Model Creation Specifications

2.3.1 Model Package Structure

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

NOTE

- The model package specifications apply when you import one model. For multiple models, such as multiple files, use custom images instead.
- 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 **Specifications for Custom Images** and **Creating a Custom Image on ECS**.
- 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 a Model Inference Code File**.
  - The .py file on which **customize_service.py** depends can be directly stored in the **model** directory. Use relative import for the custom package.
  - The other files on which **customize_service.py** depends can be stored in the **model** directory. Use absolute paths to access these files. For details, see **Obtaining an Absolute Path**.

ModelArts provides samples and sample code for multiple engines. You can edit 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 or directory name
├── ocr
|   ├── model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   ├── <<Custom Python package>> (Optional) Your Python package, which can be directly referenced in model inference code
|   |   ├── saved_model.pb (Mandatory) Protocol buffer file, which contains the graph 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 a *.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 allowed.
|   |   ├──customize_service.py (Mandatory) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file is allowed.
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 or directory name
├── resnet
|   ├── model (Mandatory) Name of a fixed subdirectory, which is used to store model-related files
|   |   ├── <<Custom Python package>> (Optional) Your 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 allowed.
|   |   ├──customize_service.py (Mandatory) Model inference code. The file name is fixed to
customize_service.py. Only one model inference code file is allowed. 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.

2.3.2 Specifications for Editing a Model Configuration File

You must edit the **config.json** file when publishing a model. The configuration file describes the model usage, computing framework, accuracy, inference code dependency package, and model API.

Configuration File Format

The configuration file is in JSON format. **Table 2-9** describes the parameters.

Table 2-9 Parameters

Parameter	Mandatory	Type	Description
model_algorithm	Yes	String	Model algorithm, which shows the model usage. The value must start with a letter and contain no more than 36 characters. Chinese characters and special characters (&!'\"<>=) are not allowed. Major model algorithms include **image_classification**, **object_detection**, and **predict_analysis**.

Parameter	Mandatory	Type	Description
model_type	Yes	String	Model AI engine, which indicates the computing framework used by a model. Major 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**, a model will be created using a custom image. In this case, the **swr_location** parameter is mandatory. For details about how to create an image, see **Specifications for Custom Images**.
runtime	No	String	Model runtime environment. Python 2.7 is used by default. The value of **runtime** depends on **model_type**. If **model_type** is set to **Image**, you do not need to set **runtime**. If **model_type** is set to a mainstream framework, select a runtime environment matching the engine. For details about the supported runtime 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*.
metrics	No	Object	Model precision information, including the F1 score, recall, precision, and accuracy. For details about the **metrics** object structure, see **Table 2-10**. The result is displayed in the model precision area on the model details page.

Parameter	Mandatory	Type	Description
apis	No	API array	Structure data of requests received and returned by a model. It is a RESTful API array provided by a model. For details about the API structure, see **Table 2-11**. For details about the code example, see **Code Example of apis Parameters**. • If **model_type** is set to **Image**, an AI application will be created using a custom image. APIs with different paths can be declared in **apis** based on the request path exposed by the image. • If **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 **/**.
dependencies	No	Dependency array	Package on which the model inference code depends, which is structure data. You must provide the package name, installation method, and version constraints. The dependency package can be installed only using **pip**. **Table 2-14** describes the dependency array. If the model package does not contain the **customize_service.py** inference code file, you do not need to set dependencies. Dependency packages cannot be installed for custom image models. **NOTE** The **dependencies** parameter accepts multiple dependency arrays in a list format. It applies to scenarios where the default installation packages have dependency relationships. The top packages are installed first. The wheel package can be used for dependency installation, and it 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?**
health	No	**health** data structure	Health check API configuration. This parameter is mandatory only when **model_type** is set to **Image**. To ensure uninterrupted services during a rolling upgrade, ModelArts requires a health check API. For details about the **health** data structure, see **Table 2-16**.

Table 2-10 metrics object

Parameter	Mandatory	Type	Description
f1	No	Number	F1 score. The value is rounded to 17 decimal places.
recall	No	Number	Recall. The value is rounded to 17 decimal places.
precision	No	Number	Precision. The value is rounded to 17 decimal places.
accuracy	No	Number	Accuracy. The value is rounded to 17 decimal places.

Table 2-11 api structure

Parameter	Mandatory	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 2-12**.
response	No	Object	Response body. For details, see **Table 2-13**.

Table 2-12 request structure

Parameter	Mandatory	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.

Parameter	Mandatory	Type	Description
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 2-13 response structure

Parameter	Mandatory	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 2-14 dependency array

Parameter	Mandatory	Type	Description
installer	Yes	String	Installation method. Only **pip** is supported.
packages	Yes	package array	Dependency package collection. For details about the package array, see **Table 2-15**.

Table 2-15 Package array

Parameter	Mandatory	Type	Description
package_name	Yes	String	Dependency package name. Chinese characters and special characters (&!'"<>=) are not allowed.
package_version	No	String	Dependency package version. Leave it blank if it is not required. 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 installed version is not earlier than the specified version. • **ATMOST** indicates that the installed version 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 2-16 Health check data structure

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.

Parameter	Mandatory	Type	Description
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 a positive integer no more than 2147483647.
failure_threshold	Yes	String	Maximum number of health check failures. Enter a positive integer 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": "application/json",
        "data": {
            "type": "object",
            "properties": {
                "mnist_result": {
                    "type": "array",
                    "item": [
                        {
                            "type": "string"
                        }
                    ]
```

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

Example of an 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 an 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 actual engine type.

- 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

```
{
    "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": "application/json",
                "data": {
                    "type": "object",
                    "properties": {
                        "mnist_result": {
                            "type": "array",
                            "item": [{
                                "type": "string"
                            }]
                        }
                    }
                }
            }
        }
    ],
    "dependencies": []
}
```

Example of a 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

```
{
    "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 a Custom Image Model Configuration File

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

- Here is an example of how to make a model prediction request for image files.

  To upload files, click the file upload button on the inference page.

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

- Here is an example of how to make a model prediction request for JSON data.

  The input parameter is of string type. To enter prediction requests, use the text box on the inference page.

  ```
  {
      "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 a 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": "application/json",
        "data": {
            "type": "object",
            "properties": {
                "resp_data": {
                    "type": "array",
                    "items": [
                        {
                            "type": "object",
                            "properties": {
                                "predict_result": {}
                            }
                        }
                    ]
                }
            }
        }
    }
  ]
}
```

Example of the 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": "application/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"
                }
            ]
        }
    ]
}
```

2.3.3 Specifications for Writing a Model Inference Code File

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 Writing 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 2-17**. The ModelArts environment has already configured the necessary Python packages for import statements, so you do not need to install them separately.

Table 2-17 Parent class and import statement of each model type

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
MindSpore	SingleNodeService	from model_service.model_service import SingleNodeService

2. The following methods can be overridden.

Table 2-18 Methods to be overridden

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. To implement model loading logic, override this method for PyTorch and Caffe-based models.
__init__(self, model_path)	Initialization method, which is suitable for models created based on machine learning frameworks. This method initializes the model path (**self.model_path**). In Spark_MLlib, this method also initializes SparkSession (**self.spark**).
_preprocess(self, data)	Preprocess method, which is called before an inference request and converts API request data into the model's expected input format.
_inference(self, data)	Inference request method. You are advised not to override this method, as it will replace the built-in inference process in ModelArts with your custom logic.
_postprocess(self, data)	Postprocess method, which is called after an inference request is complete and converts the model output to the API output.

NOTE

- You can override the preprocess and postprocess methods for preprocessing the API input and postprocessing the inference output.
- Overriding the init method of the parent model class may cause a model to run abnormally.

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

NOTE

The inference code requires an absolute file path for reading files. You can obtain the local path to the model from the **self.model_path** attribute.

- When TensorFlow, Caffe, or MXNet is used, **self.model_path** indicates the path to the model file. The following provides an example:

```
# Reads the label.json file in the model directory.
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 to the model file. The following provides an example:

```
# Reads the label.json file in the model directory.
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. The API accepts data in either **multipart/form-data** or **application/json** format for pre-processing, actual inference, and post-processing.

- **multipart/form-data** request

```
curl -X POST \
  <modelarts-inference-endpoint> \
  -F image1=@cat.jpg \
  -F images2=@horse.jpg
```

The 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 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 **Tensorflow2.1**.

- 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 sample code 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.

Inference Script Example of Custom Inference Logic

Customize a dependency package in the configuration file by referring to **Example of the 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

Python logging used by inference base images allows the display of only warning logs. To query INFO logs, set the log level to INFO in the code.

```
# -*- 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 HTTPS request formats
    # 1. Request in form-data format: data = {"Request key value":{"File name":<File io>}}
    # 2. Request in JSON format: data = json.loads("JSON body passed in 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 multiple models or models that are not supported by ModelArts, 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)

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

2.3.4 Specifications for Using a Custom Engine to Create a Model

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

Before deploying such a model as a service, ModelArts downloads the SWR image to the cluster and starts the image as a container as the user whose UID is 1000 and GID is 100. Then, ModelArts downloads the OBS file to the **/home/mind/model** directory in the container and runs the boot command preset in the SWR image. ModelArts registers an inference API with API Gateway for you to access the service.

Specifications for Using a Custom Engine to Create a Model

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

- SWR image specifications
  - A common user named **ma-user** in group **ma-group** must be built in the SWR image. Additionally, the UID and GID of the user must be 1000 and 100, respectively. The following is the dockerfile command for the built-in user:
    ```
    groupadd -g 100 ma-group && useradd -d /home/ma-user -m -u 1000 -g 100 -s /bin/bash ma-user
    ```
  - Specify a command for starting the image. In the dockerfile, specify **cmd**. The following shows an example:
    ```
    CMD sh /home/mind/run.sh
    ```
    Customize the boot file **run.sh**. The following is an example:
    ```
    #!/bin/bash

    # User-defined script content
    ...

    # run.sh calls app.py to start the server. For details about app.py, see "HTTPS Example".
    python app.py
    ```


> **NOTE**
>
> You can also customize the boot command for starting an image. Enter the customized command during model creation.

  - The provided service can use the HTTPS/HTTP protocol and listening container port. Set the protocol and port number based on the inference API defined by the model. For details about the HTTPS protocol, see **HTTPS Example**.
  - (Optional) On port provided by the service for external access, enable health check with URL **/health**. (The health check URL must be **/health**.)
- OBS model package specifications

  The name of the model package must be **model**. For details about the model package specifications, see **Model Package Specifications**.
- File size specifications

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

HTTPS Example

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

```
from flask import Flask, request
import json

app = Flask(__name__)

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

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


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

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

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

Debugging on a Local Computer

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

1. Download the custom image, for example, **custom_engine:v1** to the local computer.
2. Copy the model package folder **model** to the local computer.
3. Run the following command in the same directory as the model package folder to start the service:
   ```
   docker run --user 1000:100 -p 8080:8080 -v model:/home/mind/model    custom_engine:v1
   ```

   NOTE

   This command is used for simulation only because the directory mounted to **-v** is assigned the root permission. In the cloud environment, after the model file is downloaded from OBS to **/home/mind/model**, the file owner will be changed to **ma-user**.
4. Start another terminal on the local computer and run the following command to obtain the expected inference result:
   ```
   curl https://127.0.0.1:8080/${Request path to the inference service}
   ```

Deployment Example

The following section describes how to use a custom engine to create a model.

1. Create a model and view model details.

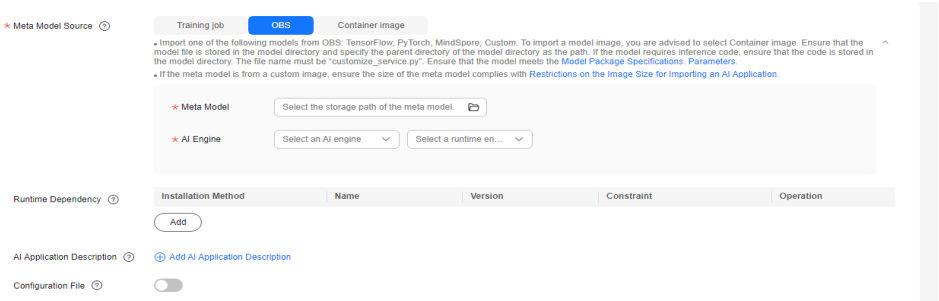
   Log in to the **ModelArts console**. Choose **Model Management** and click **Create Model**. On the page that is displayed, set the following parameters:

   - **Meta Model Source**: Select **OBS**.
   - **Meta Model**: Select a model package from OBS.
   - **AI Engine**: Select **Custom**.
   - **Engine Package**: Select a container image from SWR.

     Retain default settings for other parameters.

   Click **Create now**. Wait until the model status changes to **Normal**.

   Figure 2-6 Creating a model

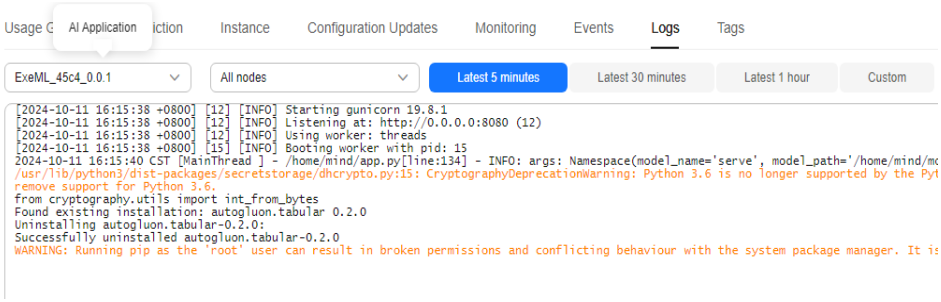
   

   Click the model name to access its details page.
2. Deploy the AI application as a service and view service details.

   On the model details page, choose **Deploy** > **Real-Time Services** in the upper right corner. On the **Deploy** page, select a proper instance flavor (for example, **CPU: 2 vCPUs 8 GB**), retain default settings for other parameters, and click **Next**. When the service status changes to **Running**, the service has been deployed.

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

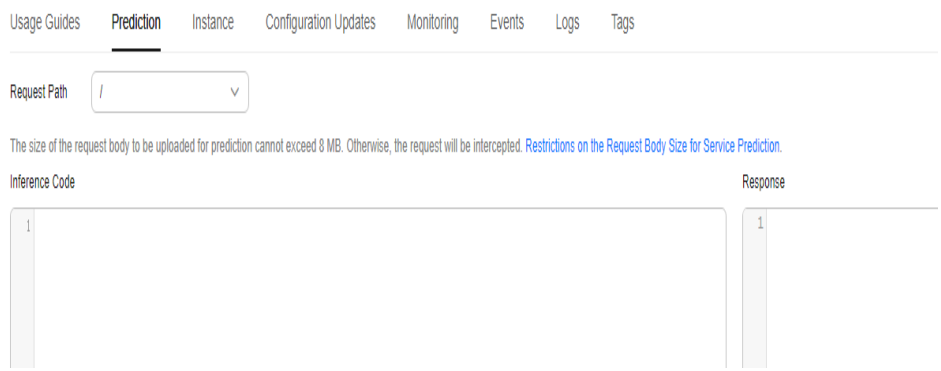
Figure 2-7 Service logs



3. Use the service for prediction.

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

Figure 2-8 Service prediction



2.3.5 Examples of Custom Scripts

To create a model in ModelArts by importing a model file from OBS, the model file package needs to comply with the ModelArts model package specifications. Additionally, the inference code and configuration file must also meet the requirements set by ModelArts.

This section provides custom script examples (including inference code examples) for common AI engines. For details about how to write model inference code, see **Specifications for Writing a Model Inference Code File**.

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 keys 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 one 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:
            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 keys 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 parent class and import statement of each model type, see **Table 2-17**. This example calls the parent class inference request method **_inference(self, data)**. The method does not need to be overridden in the following code.

```
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

    # 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
```

Tensorflow2.1

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 which inherits properties from its parent model class. For details about the parent class and import statement of each model type, see **Table 2-17**.

```
import logging
import threading

import numpy as np
```

```
import tensorflow as tf
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())
        }
```


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 which inherits properties from its parent model class. For details about the parent class and import statement of each model type, see **Table 2-17**.

```
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)),
    # Convert data to 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 custom function to load the model.
        self.model = Mnist(model_path)
         # Load labels.
        self.label = [0,1,2,3,4,5,6,7,8,9]
        # Labels can also be loaded by label file.
        # Reads the label.json file in the model directory.
        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)
```



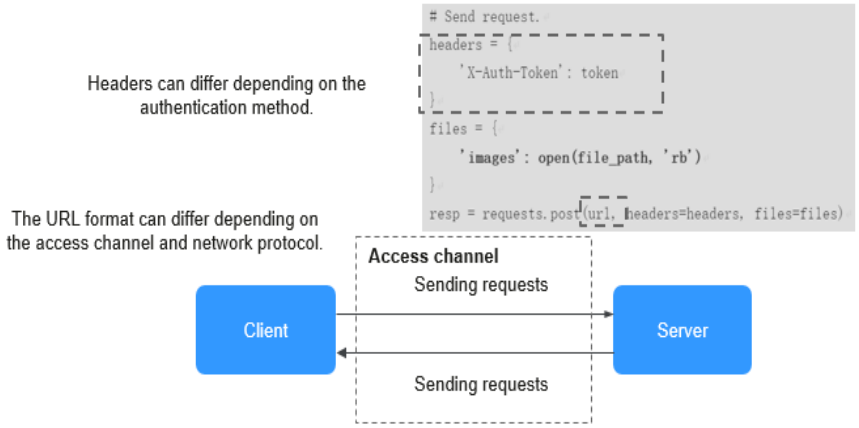
```
    def _preprocess(self, data):

        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)
    # Set the model to evaluation mode.
    model.eval()

    return model
```


2.4 Deploying a Model as Real-Time Inference Jobs

2.4.1 Deploying and Using Real-Time Inference

After creating a model, you can deploy it as a real-time service. If a real-time service is in the Running status, it has been deployed. This service provides a standard, callable RESTful API. When accessing a real-time service, you can choose the authentication method, access channel, and transmission protocol that best suit your needs. These three elements make up your access requests and can be mixed and matched without any interference. For example, you can use different authentication methods for different access channels and transmission protocols.

Figure 2-9 Authentication method, access channel, and transmission protocol



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

- **Token-based authentication**: The validity period of a token is 24 hours. When using a token for authentication, cache it to prevent frequent calls.
- **AK/SK-based authentication**: AK/SK is used to sign requests and the signature is then added to the request for authentication. 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.
- **App authentication**: Add a parameter to the request header to complete the authentication. The authentication is simple and permanently valid.

ModelArts allows you to call APIs to access real-time services in the following ways (HTTPS requests are used as examples):

- **Accessing a Real-Time Service Through a Public Network**: By default, ModelArts inference uses the public network to access real-time services. A standard, callable RESTful API is provided after deployment of a real-time service.
- **Accessing a Real-Time Service Through a VPC High-Speed Channel**: When using VPC peering for high-speed access, your service requests are sent

directly to instances via VPC peering, bypassing the inference platform. This results in faster service access.

Real-time service APIs are accessed using HTTPS by default. Additionally, the following transmission protocols are also supported:

- **Accessing a Real-Time Service Using WebSocket**: 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 the server is successful, a persistent connection will be established between them and data can be transferred bidirectionally.
- **Accessing a Real-Time Service Using Server-Sent Events**: Server-Sent Events (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.

2.4.2 Deploying a Model as a Real-Time Service

Real-time inference involves taking user inputs or queries via the internet and instantly providing processed outcomes or choices using AI or machine learning models hosted on remote servers or cloud platforms. Real-time inference uses cloud-based models to deliver fast and reliable AI services. It offers powerful analysis, predictions, and understanding without needing local model deployments. This solution works best for tasks requiring quick responses and interactions.

ModelArts allows you to deploy a model as a web service that provides a real-time test UI and monitoring capabilities. The deployed real-time service offers an accessible API for predictions and calls. Real-time inference is used in situations that need fast responses, like online intelligent customer service and autonomous driving decisions.

This section describes how to deploy your model as a real-time service on ModelArts and use it for predictions.

Billing

Deploying a service in ModelArts uses compute and storage resources, which are billed. Compute resources are billed for running the inference service. Storage resources are billed for storing data in OBS. For details, see **Table 2-19**.

Table 2-19 Billing items

Billing Item		Description	Billing Mode	Billing Formula
Compute resources	Public resource pool	Usage of compute resources. For details, see **ModelArts Pricing Details**.	Pay-per-use	Specification unit price x Number of compute nodes x Usage duration

Billing Item		Description	Billing Mode	Billing Formula
	Dedicated resource pool	Fees for dedicated resource pools are paid upfront upon purchase. There are no additional charges for service deployment. For details about dedicated resource pool fees, see **Dedicated Resource Pool Billing Items**.	N/A	N/A
Event notification (billed only when enabled)		This function uses Simple Message Notification (SMN) to send a message to you when the event you selected occurs. To use this function, enable event notification when creating a training job. For pricing details, see **SMN Pricing Details**.	Pay by actual usage	• SMS: SMS notifications • Email: Email notifications + Downstream Internet traffic • HTTP or HTTPS: HTTP or HTTPS notifications + Downstream Internet traffic
Run logs (billed only when enabled)		Log Tank Service (LTS) collects, analyzes, and stores logs. If **Runtime Log Output** is enabled during service deployment, you will be billed if the log data exceeds the LTS free quota. For details, see **Log Tank Service Pricing Details**.	Pay by actual log size	After the free quota is exceeded, you are billed based on the actual log volume and retention duration.

Constraints

A user can create up to 20 real-time services.

Prerequisites

- A ModelArts model in the **Normal** state is available. For details about how to create models, see **Creating a Model**.
- The account is not in arrears to ensure available resources for service running.
- To mount SFS Turbo to a real-time service, first create an SFS Turbo file system and associate it with the service. Follow these steps:
  a. Create an SFS Turbo file system. For details, see **Create a File System**.

b. On the **Standard Cluster** page, click the resource pool where you want to deploy the service. Copy the value of the **Network** field and exit the details page

c. Exit the details page. Click the **Network** tab and search for the target network using the copied information. Click **Interconnect VPC**, select the VPC and subnet where your SFS Turbo is located, and click **OK**.

Alternatively, choose **More** > **Add sfsturbo** and select the SFS Turbo file system you want to mount. In this step, the ECS specifications of the SFS Turbo file system must support multiple NICs. Otherwise, attaching NICs fails.

Deploying a Real-Time Service (Synchronous Request)

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**.
2. In the real-time service list, click **Deploy** in the upper left corner.
3. Configure parameters.

   a. Configure basic parameters. For details, see **Table 2-20**.

   Table 2-20 Basic parameters

Parameter	Description
Name	Name of a real-time service.
Auto Stop	Time for your service to automatically stop running. This helps you avoid unnecessary billing. If you disable this feature, your real-time service will continue running and you will be billed accordingly. By default, this feature is enabled and set to stop the service 1 hour after it starts. The options are **1 hour**, **2 hours**, **4 hours**, **6 hours**, and **Custom**. If you select **Custom**, you can enter any integer from 1 to 24.
Description	Brief description for a real-time service.

   b. Enter key information including the resource pool and model configurations. For details, see **Table 2-21**.

Table 2-21 Parameters

Parameter	Sub-Parameter	Description
Resource Pool	Public Resource Pool	Public resource pool for deploying the real-time service. Public resource pools provide large-scale public computing clusters, which are allocated based on job parameter settings. Resources are isolated by job. CPU/GPU resource pools are available for you to select. The pricing for resource pools varies depending on their flavors. For details, see **Product Pricing Details**. Public resource pools only support the pay-per-use billing mode.
	Dedicated Resource Pool	Dedicated resource pool for deploying the real-time service. The resources provided in a dedicated resource pool are exclusive and more controllable. Select a dedicated resource pool flavor. The physical pools with logical subpools created are not supported temporarily.
Model and Configuration	Model Source	Model source for deploying the real-time service. Choose **My Model** or **My Subscriptions** as needed. • **My Model**: Models either trained on ModelArts or developed elsewhere and then uploaded to ModelArts. • **My Subscriptions**: Models subscribed from AI Gallery.
	Model and Version	Model and version that are in the **Normal** state.
	Traffic Ratio (%)	Traffic percentage of the current model version. Service call requests are allocated to the current version based on this proportion. If you deploy only one version of a model, 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%**.

Parameter	Sub-Parameter	Description
	Instance Flavor	Instance flavor of the real-time service to ensure smooth operation. Select available flavors based on the list displayed on the console. The flavors in gray cannot be used in the current environment. If no public resource pool flavors are available, use a dedicated resource pool. When deploying the service with the selected flavor, there will be necessary system consumptions. This means that the actual resources required will be greater than the flavor.
	Instances	Number of instances for the current model version. If you set the number of instances to **1**, the standalone computing mode is used. If you set the number of instances to a value greater than **1**, the distributed computing mode is used. Select a computing mode based on your actual needs.
	Environment Variable	Environment variables you need to inject 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 Model and Configuration	If the selected model has multiple versions, you can add multiple versions and configure traffic ratios for each version. You can use gray release to smoothly upgrade the model version. Free compute specifications do not support gray release of multiple versions.

Parameter	Sub-Parameter	Description
	Mount Storage	This parameter is displayed when the resource pool is a dedicated resource pool. This feature will mount a storage volume to compute nodes (instances) as a local directory when the service is running. This is a good option to consider when dealing with large input data or models. **SFS Turbo** Preparations for mounting SFS Turbo: Storage mounting is allowed only for services deployed in a dedicated resource pool which has interconnected with a VPC or associated with SFS Turbo. • To interconnect a VPC is to interconnect the VPC where SFS Turbo belongs to a dedicated resource pool network. For details, see **Interconnect with a VPC**. • You can associate HPC SFS Turbo file systems with dedicated resource pool networks. Parameters: • **File System Name**: Select the target SFS Turbo file system. A cross-region SFS Turbo file system cannot be selected. • **Mount Path**: Enter the mount path of the container, for example, **/sfs-turbo-mount/**. Select a new directory. If you select an existing directory, any existing files within it will be replaced. Notes: • A file system can be mounted only once and to only one path. Each mount path must be unique. A maximum of 8 disks can be mounted to a training job. • If you need to mount multiple file systems, do not use same or similar paths, for example, **/obs-mount/** and **/obs-mount/tmp/**. • Once you have chosen SFS Turbo, avoid deleting the interconnected VPC or disassociating SFS Turbo. Otherwise, mounting will not be possible. When you mount the backend OBS storage on the SFS Turbo page, make sure to set the client's umask permission to 777 for normal use.

Parameter	Sub-Parameter	Description
Priority	N/A	This function is supported only for dedicated resource pools, including logical resource pools, new physical resource pools, and logical subpools. Existing physical resource pools do not support this function. You can set this parameter to preferentially schedule high-priority services. Priority values range from **1** (lowest) to **3** (highest). When training and inference jobs share the same pool, and **Preemption** is enabled on the training job creation page (For details, see **Creating a Training Job**), an inference task with a higher priority can preempt a training job with a lower priority.
Traffic Limit	N/A	Maximum number of times a service can be accessed within a second. You can configure this parameter as needed.
WebSocket	N/A	Specifies whether to deploy a real-time service as a WebSocket service. Change the communication protocol of the service from HTTP/HTTPS to WebSocket. WebSocket enables bidirectional, instant communication between clients and servers, making it ideal for applications like real-time predictions and chatbot interactions. Once the protocol switches to WebSocket, the service's API URL updates to a WebSocket address. Clients can then link to the service and share data using a WebSocket client. Constraints: This feature is supported only if the model is WebSocket-compliant and comes from a container image. After this feature is enabled, **Traffic Limit** and **Data Collection** cannot be set. This parameter cannot be modified after the service is deployed. For details about WebSocket real-time services, see **Full-Process Development of WebSocket Real-Time Services**.

Parameter	Sub-Parameter	Description
Application Authentication	Application	Specifies whether to control access to the real-time service through app authentication. App authentication verifies a client's identity using their AppCode and AppSecret. It allows only authorized apps to access service APIs. App authentication allows better access control and boosts service security. This feature is disabled by default. To enable this feature, see **Accessing a Real-Time Service Through App Authentication** for details and configure parameters as required.

c. (Optional) Configure advanced settings.

Table 2-22 Advanced settings

Parameter	Description
Auto Restart	Specifies whether to automatically restart a service instance when a fault occurs. After this function is enabled, the system automatically redeploys the real-time service when detecting that the real-time service is abnormal. For details, see **Configuring Auto Restart upon a Real-Time Service Fault**. Auto restart boosts service reliability, minimizes downtime, and handles hardware failures efficiently. Use this function for tasks needing reliable and stable performance.
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. 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. Custom tags are available only to the service resources of the user who has created the tags. For details about how to use tags, see **Using TMS Tags to Manage Resources by Group**

4. After confirming the entered information, deploy the 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.

You can go to the real-time service list to check if the deployment is complete. Once the service status changes from **Deploying** to **Running**, the service is deployed.

Once a real-time service is deployed, it will start immediately.

Testing Real-Time Service Prediction

After a model is deployed as a real-time service, you can debug code or add files for testing in the **Prediction** tab. Due to the limitation of API Gateway, the duration of a single prediction cannot exceed 40s.

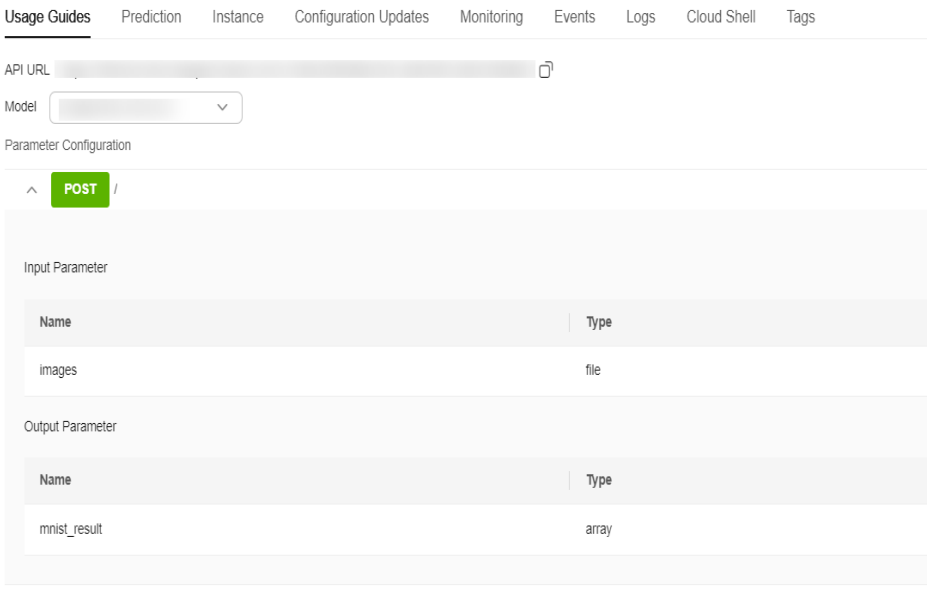
This feature is used for commissioning. Use API calling for actual production. You can select **Accessing a Real-Time Service Through Token-based Authentication**, **Accessing a Real-Time Service Through AK/SK-based Authentication**, or **Accessing a Real-Time Service Through App Authentication** based on the authentication method.

You can test the service in two ways, depending on the input request defined by the model – either by using a JSON text or a file.

- **JSON text prediction**: If the input of the deployed model is JSON text, you can enter JSON code in the **Prediction** tab for testing.
- **File Prediction**: If your model uses files as input, you can add images, audios, or videos into the **Prediction** tab to test the service.
  - The size of an input image must be less than 8 MB.
  - The maximum size of a 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 you use Ascend flavors for service deployment, you cannot predict transparent .png images because Ascend only supports RGB-3 images.

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

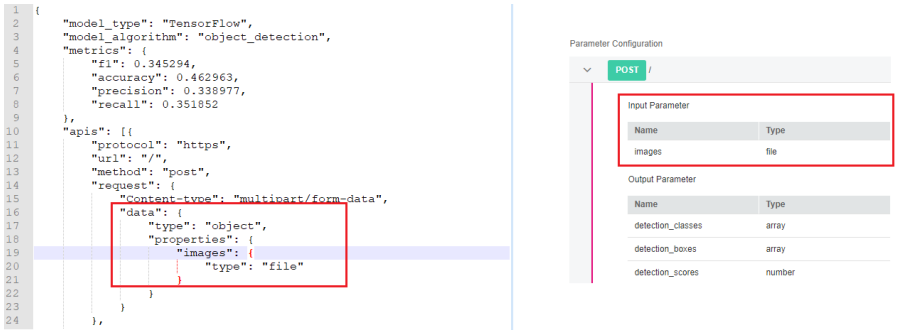
Figure 2-10 Usage Guides



The input parameters displayed in the **Usage Guides** tab depend on the model source that you select.

- If your meta model comes from a built-in algorithm, the input and output parameters are defined by ModelArts. For details, see the **Usage Guides** tab. In the **Prediction** tab, enter the corresponding JSON text or file for service testing.
- If you use a custom meta model and your own inference code and configuration file (see **Specifications for Writing the Model Configuration File**), the **Usage Guides** tab will only display your configuration file. The following figure shows the mapping between the input parameters in the **Usage Guides** tab and the configuration file.

Figure 2-11 Mapping between the configuration file and Usage Guides



The prediction methods for different input requests are as follows:

- **JSON Text Prediction**

  a. Log in to the **ModelArts console** and choose **Model Deployment** > **Real-Time Services**.

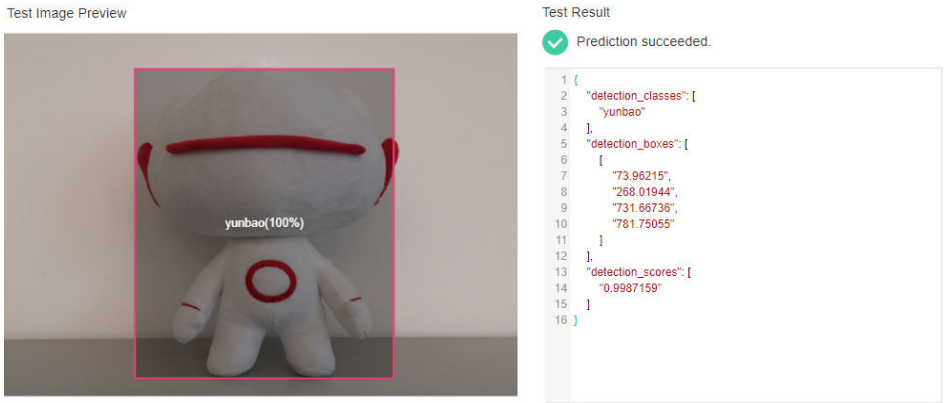
b. Click the name of the target service to access its details page. Enter the inference code in the **Prediction** tab, and click **Predict** to perform prediction.

- **File Prediction**

  a. Log in to the **ModelArts console** and choose **Model Deployment** > **Real-Time Services**.

  b. Click the name of the target service to access its details page. In the **Prediction** tab, click **Upload** and select a test file. After the file is uploaded, click **Predict** to perform a prediction test. In **Figure 2-12**, the label, position coordinates, and confidence score are displayed.

  Figure 2-12 Image prediction



Using Cloud Shell to Debug a Real-Time Service Instance Container

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

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.

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

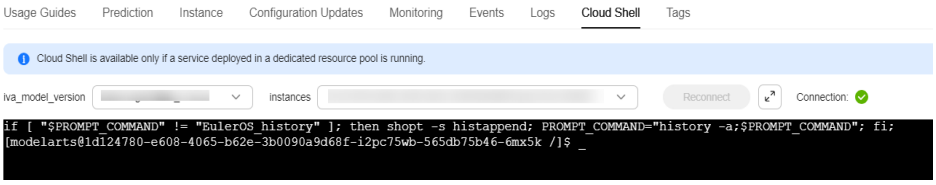
Step 2 On the real-time service list page, click the name or ID of the target service.

Step 3 Click the **Cloud Shell** tab and select the target model version and compute node.

When the connection status changes to , you have logged in to the instance container.

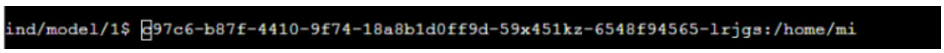
If the server disconnects due to an error or remains idle for 10 minutes, you can select **Reconnect** to regain access to the pod.

Figure 2-13 Cloud Shell



If you encounter a path display issue when logging in to Cloud Shell, press **Enter** to resolve the problem.

Figure 2-14 Path display issue



Step 4 After logging in to the container, execute the necessary debugging commands in its terminal. Example:

View logs:

```
tail -f /var/log/app.log
```

Check the service status:

```
systemctl status app
```

Run a custom script:

```
./debug_script.sh
```

Step 5 After the debugging, exit the container:

```
exit
```

After returning to the Cloud Shell terminal, you can view the debugging result or log file.

----End

FAQs

- Service deployment failed

  Alarm Status of a Deployed Real-Time Service

  Error Occurred When a Custom Image Model Is Deployed as a Real-Time Service

  Service Is Consistently Being Deployed
- Service prediction failed

  Service Prediction Failed

  Error "APIG.XXXX" Occurred in a Prediction Failure

2.4.3 Authentication Methods for Accessing Real-time Services

2.4.3.1 Accessing a Real-Time Service Through Token-based Authentication

A token specifies certain permissions in a computer system. During token-based authentication, the token is added to requests to get permissions for calling the API.

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. You can use token-based authentication to verify your identity when calling a real-time service API.

ModelArts uses tokens for accessing real-time services. Tokens typically last 24 hours. Once expired, you must get a new one. This method suits fast development and testing due to its simplicity, though tokens expire quickly.

Test the real-time service's RESTful API before deploying it to production. Send an inference request using token-based authentication with these steps:

- **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**.

Constraints

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.

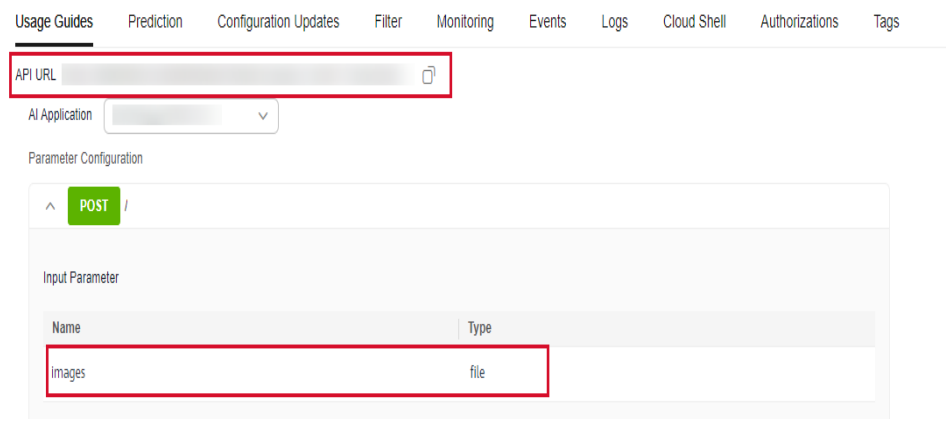
Prerequisites

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**.

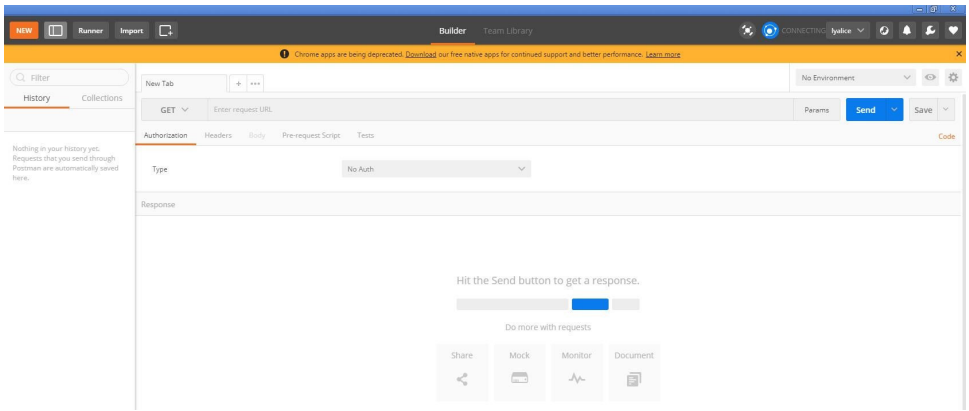
Figure 2-15 Obtaining the API URL and file prediction input parameters of a real-time service



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 2-16** shows the Postman interface.

Figure 2-16 Postman interface

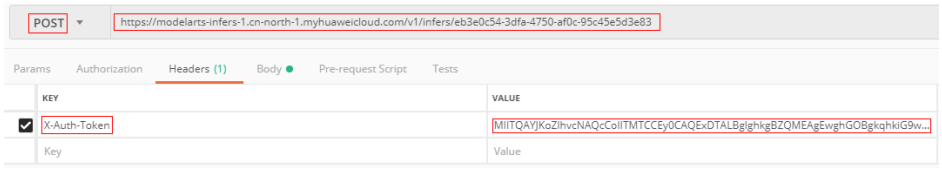


3. Set parameters on Postman. The following uses image classification as an example.
   - Select a POST task and copy the API URL to the POST text box. On the **Headers** tab page, set **Key** to **X-Auth-Token** and **Value** to the user token.

     NOTE

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

Figure 2-17 Parameter settings

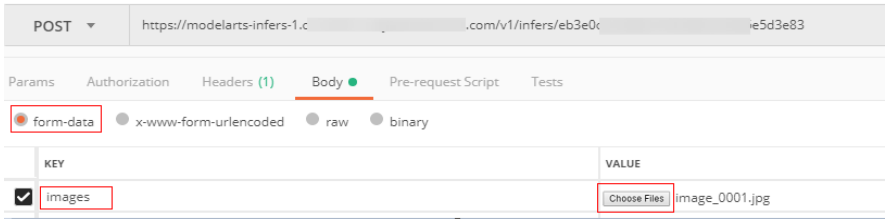


- In the **Body** tab, file input and text input are available.

  - **File input**

    Select **form-data**. Set **KEY** to the input parameter of the model, 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 2-18**.

    Figure 2-18 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 you call an API, the system provides a UUID. When the inference result is returned, 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 are determined by the model, such as **sepal_length** and **sepal_width** in this example.

4. After setting the parameters, click **send** to send the request. The result will be displayed in **Response**.
   - Inference result using file input: **Figure 2-19** shows an example. The field values in the return result vary with the model.
   - Inference result using text input: **Figure 2-20** 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 model, for example, **sepal_length** and **predictresult** in this example.

Figure 2-19 File inference result

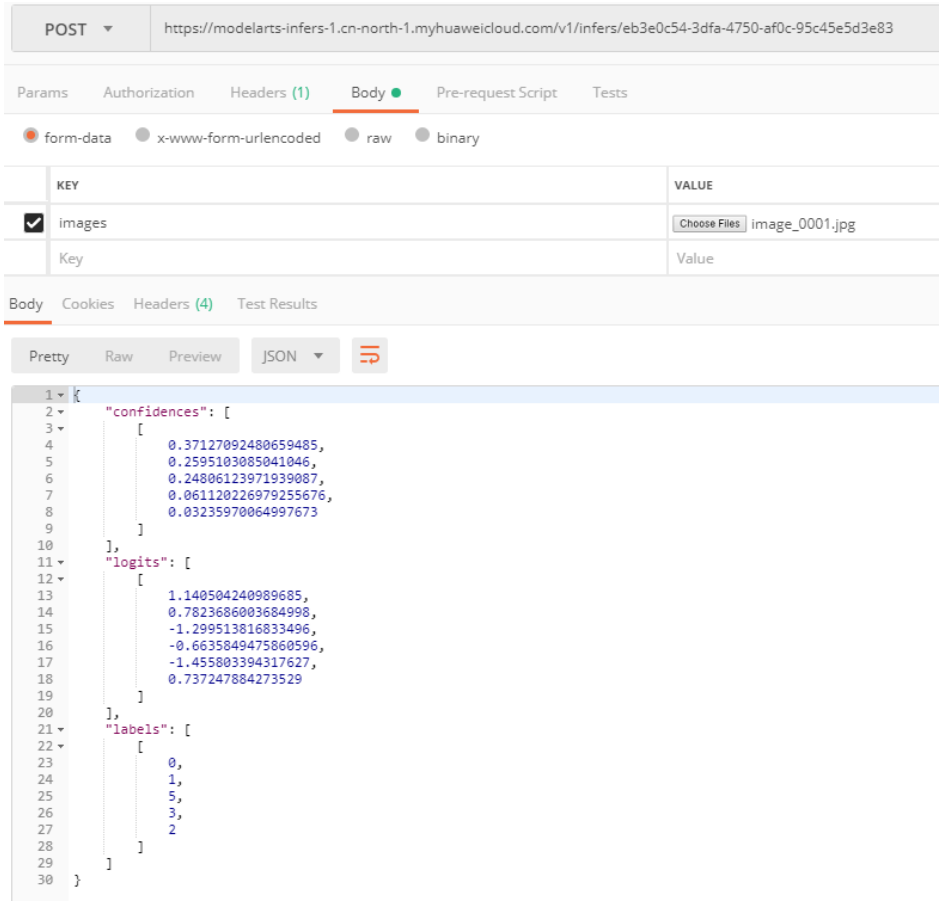
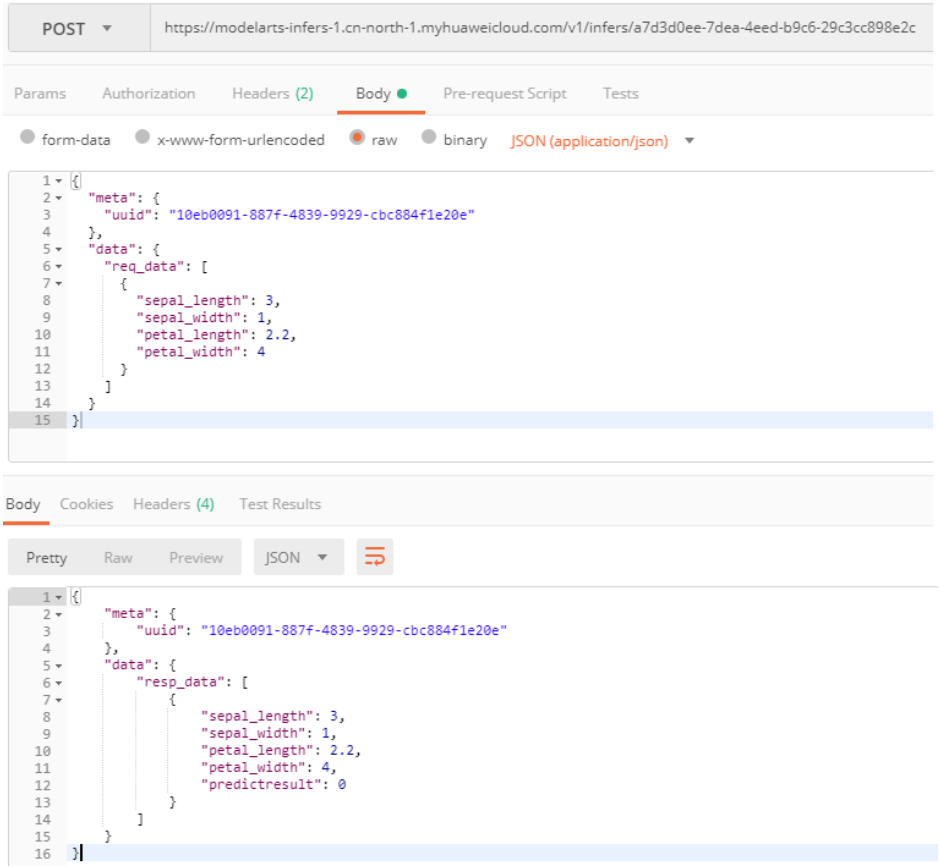


Figure 2-20 Text inference result



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

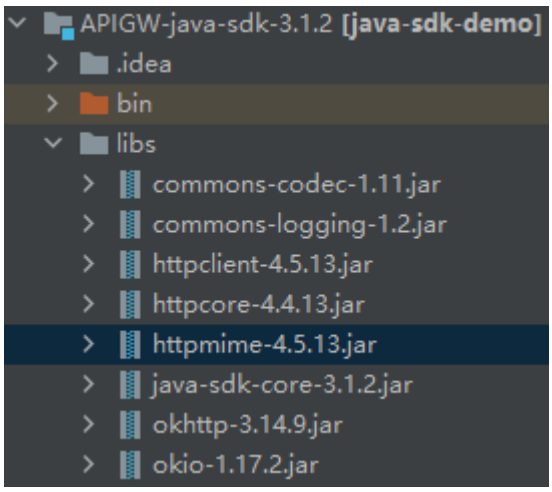
parameter of the string type. The input parameter **images** obtained in **Prerequisites** is an example. The value of **base64_data** in **body** is of the string type.

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 2-21** 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 2-21 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 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:

```
// 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.

2.4.3.2 Accessing a Real-Time Service Through AK/SK-based Authentication

AK/SK-based authentication is a method used by ModelArts for identity authentication. Access Key (AK) and Secret Key (SK) are keys used to access Huawei Cloud services.

If a real-time service is in the **Running** state, it has been deployed. This service provides a standard, callable RESTful API. You can call the API using AK/SK-based authentication.

AK/SK-based authentication supports API requests with a body not larger than 12 MB. For API requests with a larger body, use token-based authentication. For details, see **Accessing a Real-Time Service Through Token-based Authentication**.

When AK/SK-based authentication is used, you can use the APIG SDK or ModelArts SDK to access the real-time service. 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. Send an inference request.
   - **Method 1: Use Python to Send an Inference Request**
   - **Method 2: Use Java to Send an Inference Request**

Constraints

- 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.
- 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**.

The file contains the username, AK, and SK.

Figure 2-22 credential.csv

	A	B	C
1	User Name	Access Key Id	Secret Access Key
2	hu[redacted]dg	QTWA[redacted]UT2QVKYUC	MFyfvK41ba2[redacted]npdUKGpownRZlmVmHc

To generate an AK/SK pair, follow these steps:

1. Register and log in to the **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**.
5. Complete the identity authentication, download the access key, and keep it secure.

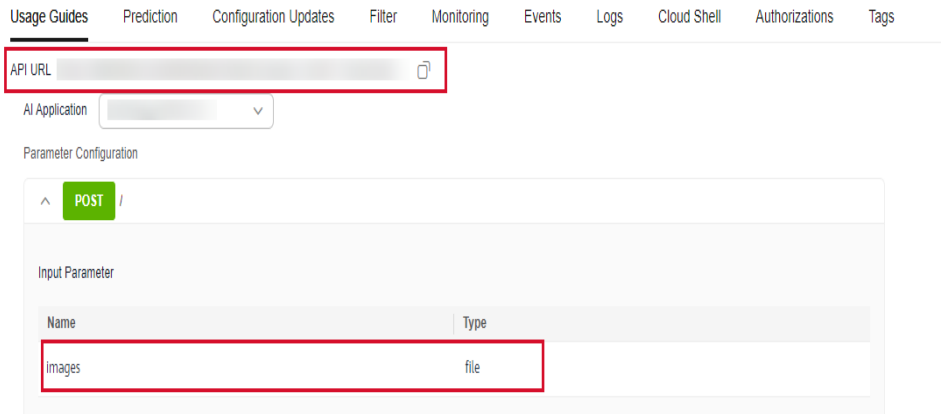
Obtaining Information About a Real-Time Service

To call an API, you will need the URL and input parameters of the real-time service. Follow these steps to obtain this information:

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**.
2. Click the name of the target service to access its details page.
3. Obtain the URL and input parameters of the service.

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

Figure 2-23 Obtaining the API URL and file 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 2-23**.

Table 2-23 Parameters of **files**

Parameter	Mandatory	Description
Request parameter	Yes	Parameter name of the real-time service.
File path	No	Path for storing the file.
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 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 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 using Base64 encoding, you need to add a decoding step to your model inference code to handle the request body.

- **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.

2.4.3.3 Accessing a Real-Time Service Through App Authentication

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 or create an application.
2. **Managing Authorization for Real-Time Services**: Manage your applications, including viewing, resetting, or deleting them, as well as binding or unbinding real-time services and obtaining your AppKey and AppSecret, or AppCode.
3. **Application Authentication**: To call an API that supports app authentication, you will need to authenticate first. There are two authentication methods: AppKey and AppSecret, or AppCode. You can choose the one that suits you best.
4. Send 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**

Constraints

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.

Prerequisites

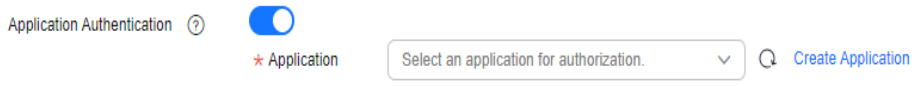
- A ModelArts model in the **Normal** state is available.
- The account is not in arrears to ensure available resources for service running.
- The local path to the inference file has been obtained. The path can be an absolute path (for example, **D:/test.png** for Windows and **/opt/data/test.png** for Linux) or a relative path (for example, **./test.png**).

Enabling Application Authentication

When deploying 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 console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**.
2. Enable application authentication.
   - When deploying a real-time service, 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, enable application authentication.

   Figure 2-24 Enabling application authentication

3. Select an application for authorization from the drop-down list. If no application is available, follow these steps to create one:

- Click **Create Application**, enter the application name and description, and click **OK**. By default, the application name is prefixed with **app_**. You can change this name if needed.
- On the **Model Deployment** > **Real-Time Services** page, click **Authorize**. On the **Manage Authorization of Real-Time Services** page, click **Create Application**. For details, see **Managing Authorization for Real-Time Services**.

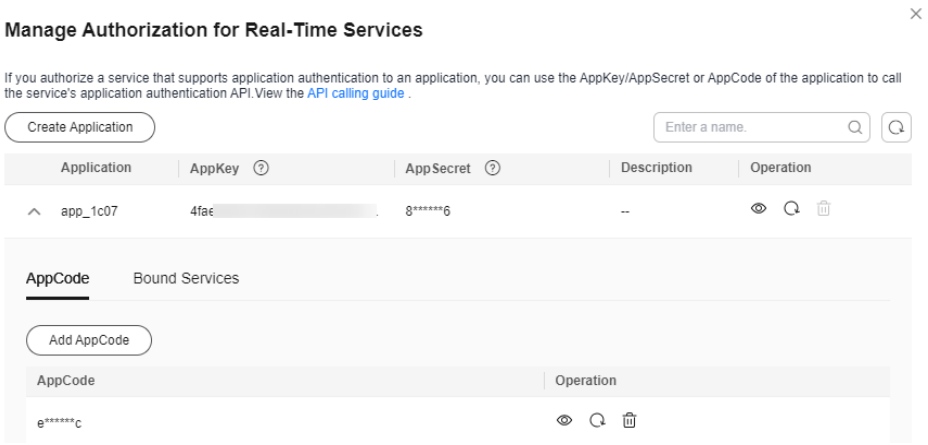
4. After enabling application authentication, authorize a service that supports application authentication to the application. Then, you can use the created AppKey/AppSecret or AppCode to call the service's API that supports application authentication.

   Once authorized, it takes 1–2 minutes for the app authentication to take effect.

Managing Authorization for Real-Time Services

If you want to use application authentication, it is good practice to create an application on the authorization management page before deploying a real-time service. In the navigation pane, choose **Model Deployment** > **Real-Time Services**. On the **Real-Time Services** page, click **Authorize**. From there, you can create, reset, or delete applications, query plaintext, unbind real-time services from applications, and obtain the AppKey/AppSecret or AppCode.

Figure 2-25 Managing authorization for real-time services



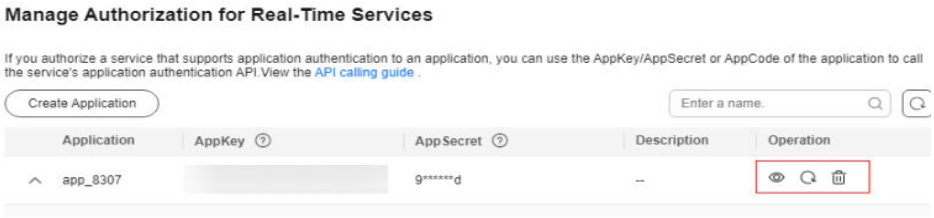
- **Creating an application**

  Click **Create Application**, enter the application name and description, and click **OK**. By default, the application name is prefixed with **app_**. You can change this name if needed.

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

  Query plaintext, 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.

Figure 2-26 Query Plaintext, Reset, or Delete



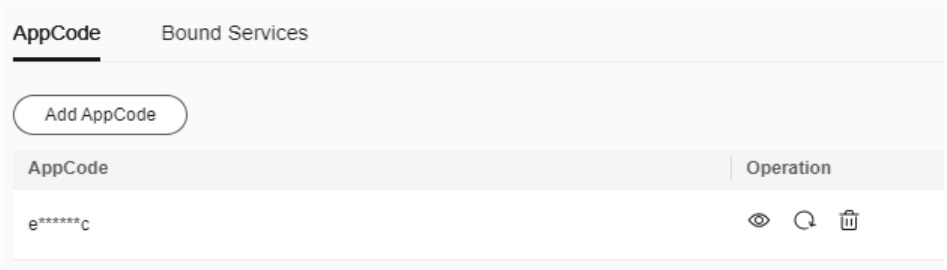
- **Unbinding a service**

  Click ⌄ next to the target application name 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 ⌄ next to the application name to show the drop-down list. Click **Add AppCode** to automatically generate an AppCode. Then, click 👁 in the **Operation** column to view the complete AppCode.

Figure 2-27 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 you use application authentication and enable simplified authentication, you can use your AppKey/AppSecret for signing and verification, or AppCode for simplified authentication. ModelArts uses simplified authentication 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.

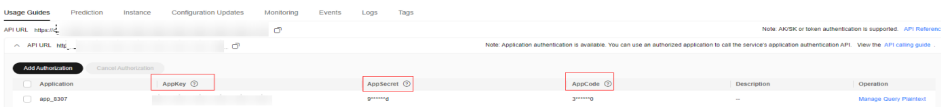
You can obtain the API URL (**url** of the real-time service), AppKey/AppSecret (**app_key** and **app_secret**), and AppCode (**app_code**) from the **Usage Guides** tab on the service details page (see **Figure 2-28**). Use the API URL for application authentication in the second line of the figure.

Modify the API URL in the following scenarios:

- If **apis** defines a path in the model configuration file, append the user-defined path to the URL, for example, *{URL of the real-time service}***/predictions/poetry**.

  If an SD WebUI inference service is deployed, add a slash (/) to the end of the calling address, for example, **https://8e******5fe.apig.******.huaweicloudexampleapis.com/v1/infers/f2682******f42/**.

Figure 2-28 Obtaining application authentication information



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.
    # API URL, for example, https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/f2682******f42, which is the URL of the real-time service in Figure 2-28.
    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 2-24**.

Table 2-24 Parameters of **files**

Parameter	Mandatory	Description
Request parameter	Yes	Parameter name of the real-time service.
File path	No	Path for storing the file.
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 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.
    # API URL, for example, "https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/
```



```
f2682******f42
    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:

```
// 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) {
      # API URL, for example, "https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/
f2682******f42
       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 appKey and appSecret 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.
        # API URL, for example, "https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/
     f2682******f42
     ```

```
    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 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.
    # API URL, for example, "https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/f2682******f42
    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)
```

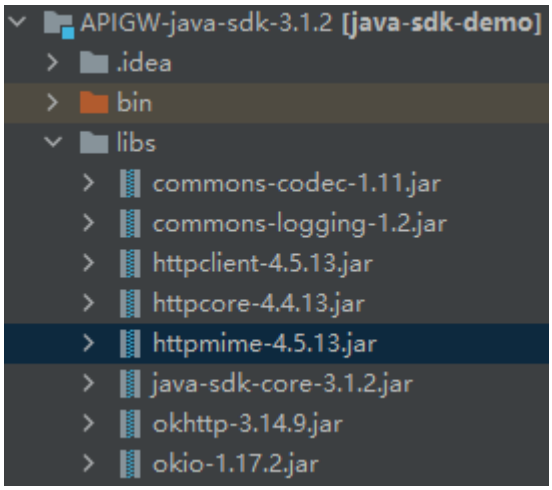
The **body** name is determined by the input parameter of the real-time service. The parameter name must be the same as that of the input parameter of the string type. **image** is used as an example. The value of **base64_data** in **body** is of the string type.

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 inference request input is in a file format, follow these steps to ensure the Java project includes the httpmime module as a dependency.
   a. Add **httpmime-x.x.x.jar** to the **libs** folder. **Figure 2-29** 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 2-29 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) {
        # API URL, for example, "https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/f2682******f42
        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 request body for reading the local inference file and performing Base64 encoding:

```
// 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) {
        # API URL, for example, "https://8e******5fe.apig.******.huaweicloudapis.com/v1/infers/f2682******f42
        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.

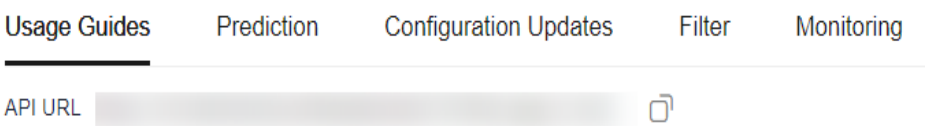
2.4.4 Accessing a Real-Time Service Through Different Channels

2.4.4.1 Accessing a Real-Time Service Through a Public Network

Context

ModelArts inference accesses real-time services through the public network by default, supporting both HTTPS and WebSocket protocols. 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.

Figure 2-30 API URL



Constraints

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.

Accessing a Real-Time Service

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

- **Accessing a Real-Time Service Through Token-based Authentication**
- **Accessing a Real-Time Service Through AK/SK-based Authentication**
- **Accessing a Real-Time Service Through App Authentication**

2.4.4.2 Accessing a Real-Time Service Through a VPC Channel

Context

To access a ModelArts real-time service from an internal VPC node of your account, you can use a VPC channel. By creating an endpoint in your VPC and connecting to the ModelArts VPC endpoint service, you can access the real-time service from your VPC endpoint. VPC access channels work best when you need strong security and fast connections, like for internal system communications.

Constraints

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.

Procedure

To access a real-time service through a VPC channel, follow these steps:

1. **Obtain the ModelArts VPC endpoint service address.**
2. **Buy and connect to a ModelArts endpoint.**
3. **Create a private DNS zone.**
4. **Access a real-time service through VPC.**

Step 1 Submit a service ticket and provide the account ID to Huawei Cloud technical support to obtain the ModelArts VPC endpoint service address.

Step 2 Buy and connect to a ModelArts endpoint.

1. Log in to the **VPC console**. In the navigation pane, choose **VPC Endpoint** > **VPC Endpoints**.
2. Click **Buy VPC Endpoint** in the upper right corner.

- **Region**: region where the VPC endpoint is located.

  Resources in different regions cannot communicate with each other. The region must be the same as that of ModelArts.
- **Service Category**: Select **Find a service by name**.
- **VPC Endpoint Service Name**: Enter the endpoint service address obtained in **step 1**. Click **Verify** on the right. The system automatically sets **VPC**, **Subnet**, and **Private IP Address**.
- **Create a Private Domain Name**: Retain the default setting.

3. Confirm the specifications, and click **Next** and then **Submit**. The VPC endpoint list page is displayed.

Step 3 Create a private DNS zone.

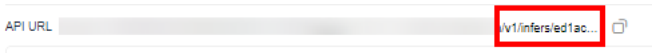
The newly created real-time service is interconnected with a dedicated gateway. The independent public domain name of ModelArts inference, that is, **infer-modelarts-***<Region ID>***.modelarts-infer.com**, is required. The intranet VPC cannot resolve the **modelarts-infer.com** domain name. You need to add private domain name resolution by referring to this step and **Step 4**.

1. Log in to the **DNS console**. In the navigation pane on the left, choose **Private Zones**.
2. Click **Create Private Zone**. Set the following parameters:
   - **Domain Name**: Enter a value in the format **infer-modelarts-***<Region ID>***.modelarts-infer.com**. Example: **infer-modelarts-cn-south-1.modelarts-infer.com**.
   - **VPC**: Select a VPC you want to associate with the private zone.
3. Click **OK**.

Step 4 Access the real-time service through the VPC.

1. Use the following API to access a real-time service through VPC:
   https://{*Private DNS domain name***}/{***URL***}**
   - *Private DNS domain name*: private domain name you set. You can also click **Access VPC** on the real-time service list page to view the domain name in the displayed dialog box.
   - *URL*: The URL for a real-time service is the part after the domain name of **API URL** in the **Usage Guides** tab of the service details page.

     Figure 2-31 Obtaining the URL

2. Use GUI-based software, cURL command, or Python to access a real-time service. For details, see **Accessing a Real-Time Service Through Token-based Authentication**.

----End

2.4.4.3 Accessing a Real-Time Service Through a 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.

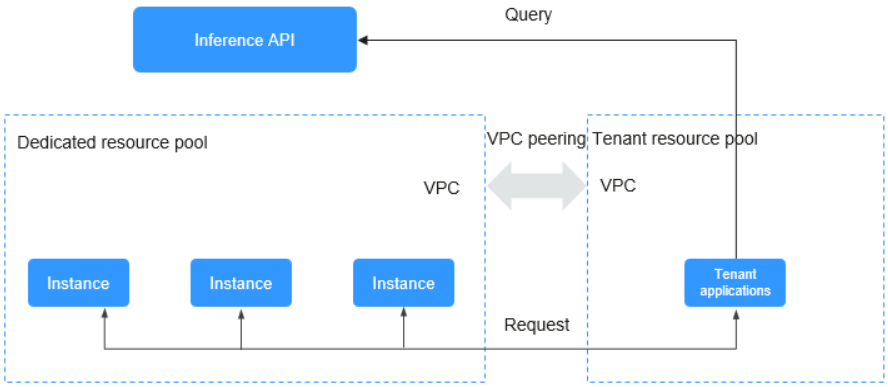
In high-speed access through VPC peering, your service requests are directly sent to instances through VPC peering but not through the inference platform. This accelerates service access. This method works best when you need high bandwidth with minimal delays, like handling real-time data or video streams.

WARNING

When using a high-speed VPC access channel for a real-time service, requests do not reach the inference platform, making these features inaccessible:

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

Figure 2-32 High-speed access through VPC peering



Constraints

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.
- 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.

Preparations

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

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 console, choose **Resource Management** > **Standard Cluster**, find the dedicated resource pool where the service is deployed, and click its name/ID to go to the resource pool details page. Obtain the network configuration. Return to the homepage, choose **Network** in the navigation pane, find the network associated with the dedicated resource pool, and interconnect your 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 2-33 Obtaining the network configuration

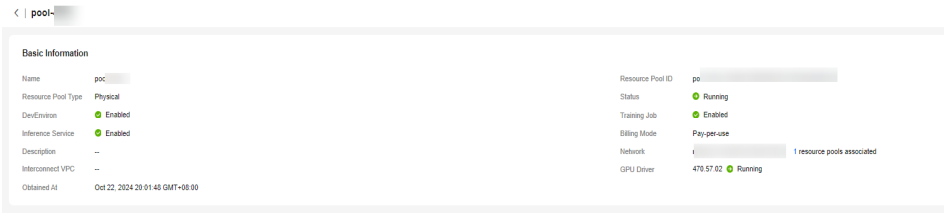


Figure 2-34 Interconnecting the VPC

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 2-35 Selecting a VPC when purchasing an ECS

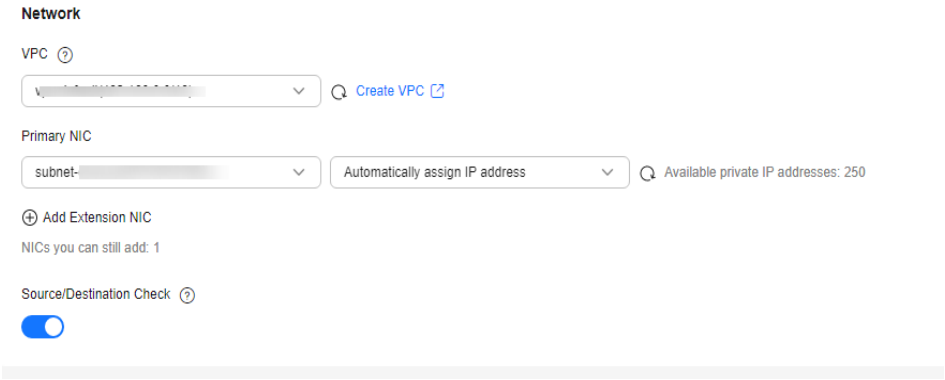
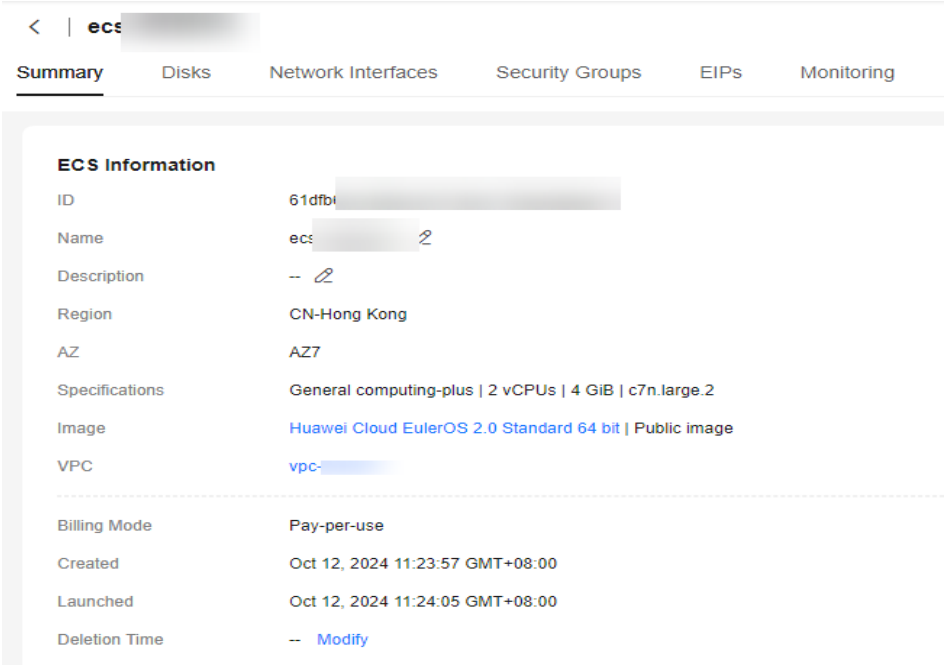


Figure 2-36 Viewing VPC information



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

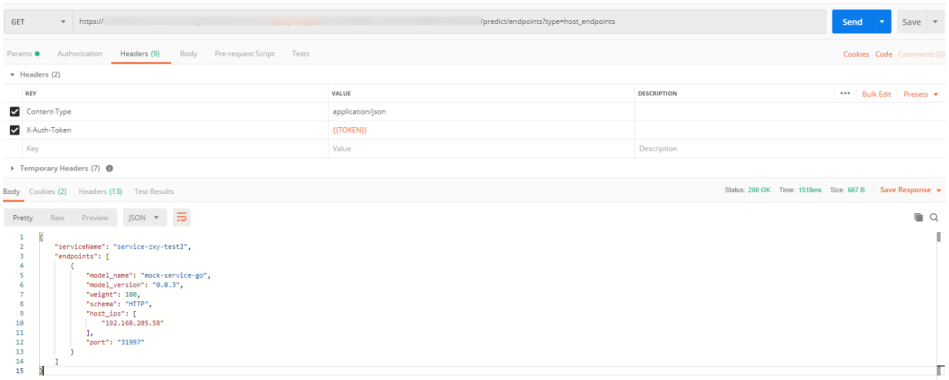
GUI software, for example, Postman can be used to obtain the IP address and port number. Alternatively, log in to the ECS, create a Python environment, and execute code to obtain the service IP address and port number.

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 2-37 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 2-38 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

2.4.5 Accessing a Real-Time Service Using Different Protocols

2.4.5.1 Accessing a Real-Time Service Using 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. WebSocket is ideal for applications requiring real-time, bidirectional communication, such as online games and real-time chat.

WebSocket can be used to deploy real-time services on ModelArts, particularly when custom images are needed to import AI models. WebSocket only supports real-time services and, when used on ModelArts, the protocol is converted to WebSocket Secure (WSS), which supports one-way authentication.

Key features of WebSocket include:

- Persistent connection: Unlike HTTP, WebSocket maintains a persistent connection between the client and server, reducing the overhead of frequent connection establishment.
- Bidirectional communication: Data can flow in both directions, allowing the server to proactively send data to the client, not just respond to client requests.
- Reduced latency and bandwidth: Real-time data transmission is possible because the connection remains open, minimizing latency and reducing unnecessary HTTP requests and bandwidth usage.

Prerequisites

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

Constraints

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

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:

- **Token-based Authentication**
- **AK/SK**
- **App Authentication**

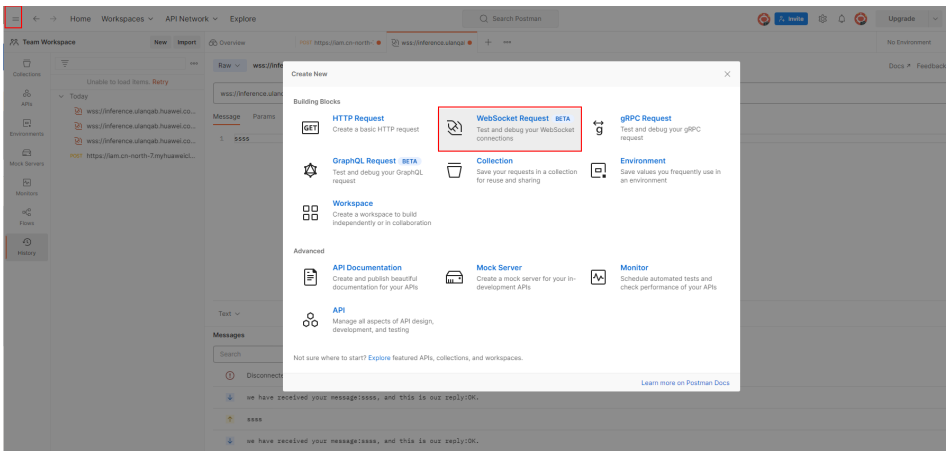
The following section uses GUI software Postman for prediction and token authentication as an example to describe how to call WebSocket.

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).

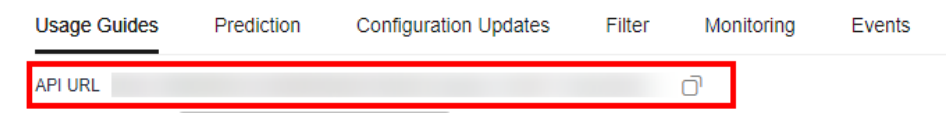
   Figure 2-39 WebSocket Request

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 2-40 Obtaining the API address

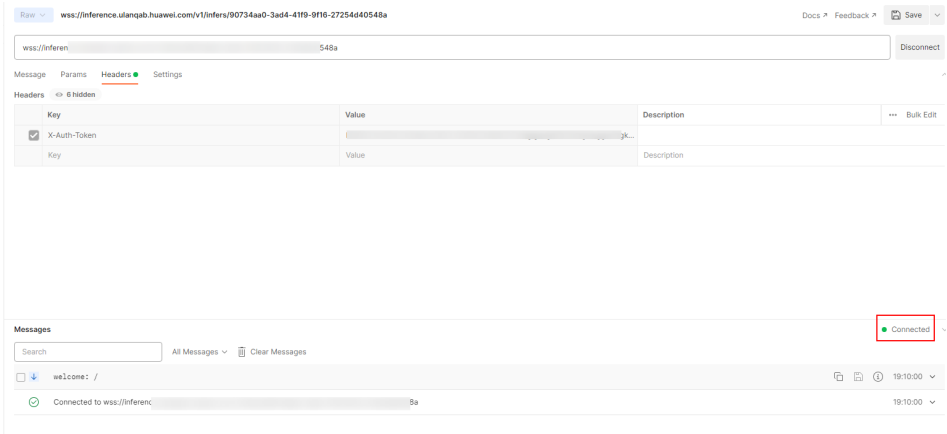


NOTE

- 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 2-41 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).

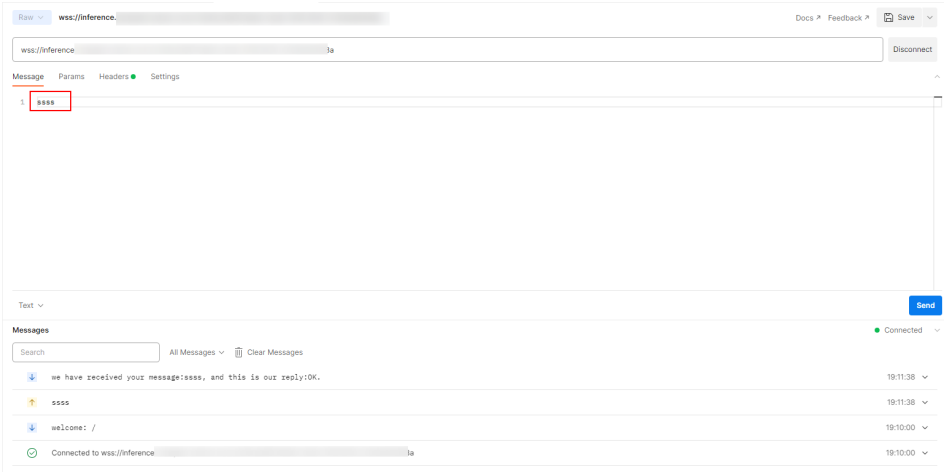
Step 2 Exchange data between the WebSocket client and the server.

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 2-42 Sending data



----End

2.4.5.2 Accessing a Real-Time Service Using 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 unidirectionally push real-time data to a client, for example, a real-time news update or stock prices.

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.

Key features of SSE include:

- Easy to use: SSE is based on the HTTP protocol and is straightforward to implement. No complex configurations or additional libraries are needed, and data can be pushed in real-time over standard HTTP connections.
- Automatic reconnection: SSE supports automatic reconnection. If the connection is interrupted, the client automatically attempts to reconnect, ensuring continuous data delivery.
- Unidirectional communication: SSE is unidirectional, meaning the server can send events to the client, but the client cannot send data back to the server through the same connection.
- Low resource usage: SSE uses HTTP connections, making it less resource-intensive compared to other real-time communication protocols like WebSocket. This makes SSE ideal for lightweight real-time data push scenarios.

Prerequisites

The image for importing the model is SSE-compliant.

Constraints

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

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:

- **Accessing a Real-Time Service Through Token-based Authentication**
- **Accessing a Real-Time Service Through AK/SK-based Authentication**
- **Accessing a Real-Time Service Through App Authentication**

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

Figure 2-43 Calling an SSE service

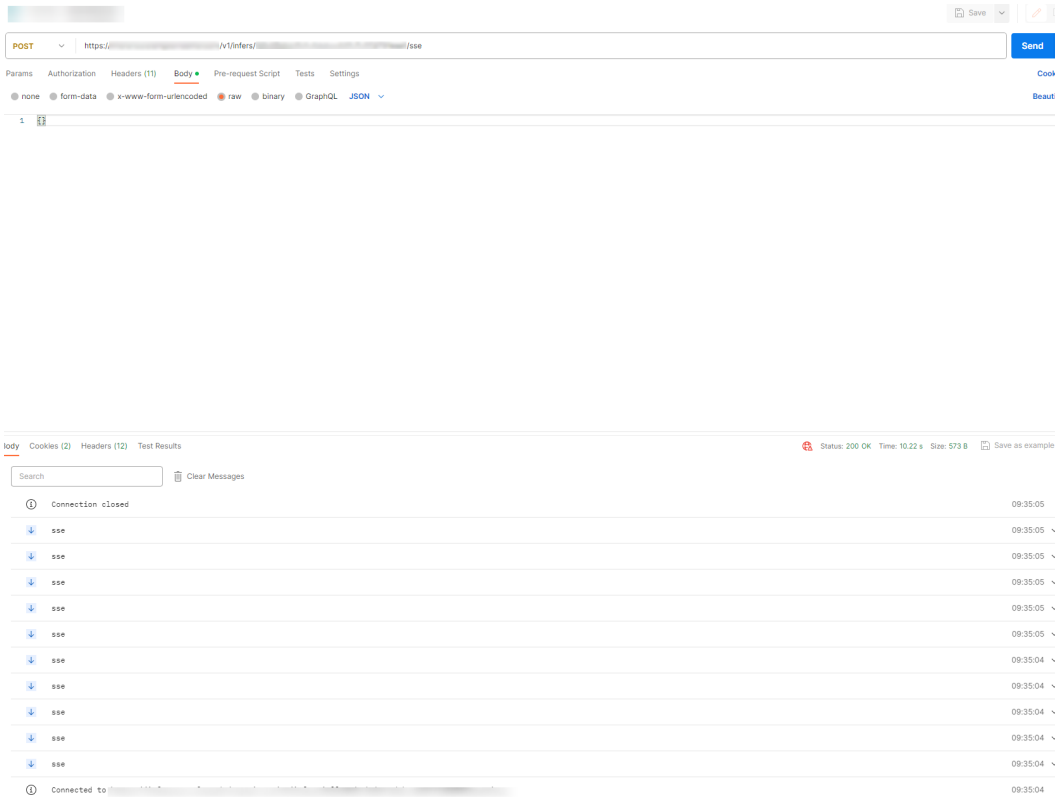


Figure 2-44 Response header **Content-Type**

NOTE

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

2.5 Deploying a Model as a Batch Inference Service

Batch inference: Processes multiple inputs at once and returns all results together. ModelArts allows you to deploy a model as a batch service. This service runs inference on batch data and stops automatically when done. Batch inference works well for offline tasks like big data analysis, batch data labeling, and model evaluation.

This section describes how to deploy a model as a batch inference service on ModelArts and how to check the prediction results.

Billing

Deploying a service in ModelArts uses compute and storage resources, which are billed. Compute resources are billed for running the inference service. Storage

resources are billed for storing data in OBS. For details, see **Inference Deployment Billing Items**.

Prerequisites

- A ModelArts model in the **Normal** state is available.
- Data to be processed in batches has been uploaded to OBS.
- At least one empty folder has been created in OBS for storing the output.

Context

- You can create up to 1,000 batch services.
- Based on the input request (JSON or file) defined by the model, different parameters are entered. If the model input is a JSON file, a configuration file is required to generate a mapping file. If the model input is a file, no mapping file is required.

Procedure

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Batch Services**.
2. Click **Deploy** in the upper left corner.
3. Configure parameters.
   a. Enter basic information, including **Name** and **Description**. A name is generated by default, for example, **service-bc0d**, which you can modify.
   b. Configure other parameters, including the resource pool and model configurations.

Table 2-25 Parameters

Parameter	Description
Resource Pool	Public Resource Pool CPU and GPU public resource pools are available for you to select. To use a public resource pool, contact the administrator to create one.
	Dedicated Resource Pool Select a specification from the resource pool specifications.
Model Source	Choose **My Model** or **My Subscriptions** as needed.
Model and Version	Select the model and version that are in the **Normal** state.

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**. • 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 model API in a batch service, and also the request path of the model service. Its value is obtained from the **url** field of **apis** in the model configuration file.
Mapping Relationship	If the model input is in JSON format, the system automatically generates the mapping based on the configuration file corresponding to the model. If the model input is other file, mapping is not required. The mapping file is generated automatically. 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 **Mapping Example**. 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 CSV data must be separated by commas (,).
Output Path	The path for storing the batch prediction results. You can select an empty folder you created.
Instance Flavor	The system provides available compute resources matching your model. Select an available resource from the drop-down list.

Parameter	Description
Instances	Number of instances for the current model 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 your actual needs.
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 feature is disabled by default. The runtime logs of batch services are stored only in the ModelArts log system. You can query the runtime logs in the **Logs** tab of the service details page. If this feature 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 LTS log management, see **Log Tank Service**. • This cannot be disabled once it is enabled. • You will be billed for log query and storage features provided by LTS. For details, see **LTS Pricing Details**. **WARNING** 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. Confirm the configurations and complete service deployment 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

   Once a batch service is deployed, it will start immediately. You will be billed for the chosen resources while it is running.

   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.

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=wZBttZj5QZrReDhz1uDzwve8GpY%3D&x-obs-security-token=gQpzb3V0aGNoaW5hixvY8V9a1SnsxmGoHYmB1SArYMyqnQT-ZaMSxHvl68kKLAy5feYvLDM..."}
```

- Requirements on the file:
  a. The file name extension must be **.manifest**.
  b. The file content must be in JSON format. Each line describes a piece of input data, which must be a specific file instead of a folder.
  c. JSON content requires a **source** field, which must be an OBS file address in either 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 obtain the path by accessing the OBS object. *<obs path>* can be **obs://** or **s3://**.
     ii. Share 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 service.

- Assume that the output path is **//test-bucket/test/**. The result is stored in the following path:

```
OBS bucket or 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.example.com:443/xgboosterdata/2.jpg?AccessKeyId=2Q0V0TQ461N26DDL18RB&Expires=1550611914&Signature=wZBttZj5QZrReDhz1uDzwve8GpY%3D&x-obs-security-token=gQpzb3V0aGNoaW5hixvY8V9a1SnsxmGoHYmB1SArYMyqnQT-ZaMSxHvl68kKLAy5feYvLDMNZWxzhBZ6Q-3HcoZMh9gISwQOVBwm4ZytB_m8sg1fL6isU7T3CnoL9jmvDGgT9VBC7dC1EyfSJrUcqfB_N0ykCsfrA1Tt_IQYZFDu_HyqVk-GunUcTVdDfWlCV3TrYcpmznZjliAnYUO89kAwCYGeRZsCsC0ePu4PHMsBvYV9gWmN9AUZIDn1sfRL4voBpwQnp6tnAgHW49y5a6hP2hCAoQ-95SpUriJ434QlymoeKfTHVMKOeZxZea-JxOvevOCGI5CcGehEJaz48sgH81UiHzl21zocNB_hpPfus2jY6KPglEJxMv6Kwmro-ZBXWuSJUDOnSYXI-3ciYjg9-h10b8W3sW1mOTFCWNGoWsd74it7l_5-7UUhoIeyPByO_REwkur2FOJsuMpGlRaPyglZxXm_jfdLFXobYtzZhbul4yWXga6oxTOkfcwykTOYH0NPoPRt5MYGYweOXXxFs3d5w2rd0y7p0QYhyTzIkk5CIz7FlWNapFISL7zdhsl8RfchTqESq94KgkeqatSF_iIvnYMW2r8P8x2k_eb6NJ7U_q5ztMbO9oWEcfr0D2f7n7Bl_nb2HIB_H9tjzKvqwngaimYhBbMRPfibvttW86GiwVP8vrC27FOn39Be9z2hSfJ_8pHej0yMlyNqZ481FQ5vWT_vFV3JHM-7I1ZB0_hIdaHfItm-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 line describes the output result of a piece of input data.
  e. The JSON 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**.

Mapping Example

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

The following shows the format of the CSV data for inference. The data must be separated by commas (,).

```
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
    }]
```



```
  }
}
```

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** state.

Step 1 Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Batch Services**.

Step 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  next to **Description** to edit the description.

Step 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 model prediction result.

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

- Prediction result file: The file is in the *xxx***.manifest** format and contains the file path and prediction result.
- Model prediction output:
  - 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**.

----End

2.6 Managing ModelArts Models

2.6.1 Viewing ModelArts Model Details

Viewing the Model List

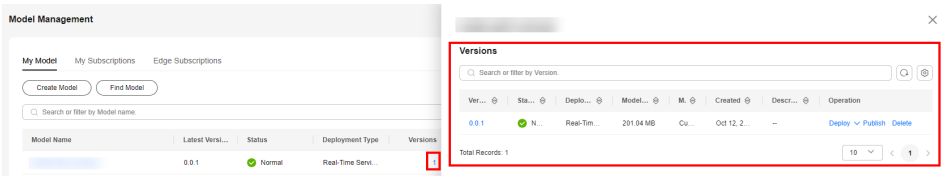
You can view all created models on the model list page. The model list page displays the following information.

Table 2-26 Model list

Parameter	Description
Model Name	Model name.
Latest Version	Latest version of a model.
Status	Model status.
Deployment Type	Types of the services that a model can be deployed as.
Versions	Number of model 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 video files. • Asynchronous request: one-off inference with results returned asynchronously (longer than 60s). This mode is suitable for real-time video inference and large videos.
Created	Model creation time.
Description	Model description.
Operation	• **Deploy**: Deploy a model as real-time services, edge services, or batch services. • **Create Version**: Create a model version. The settings of the last version are used by default, except for the version. You can change the parameter settings. • **Delete**: Delete a model. **NOTE** If a model version has been deployed as a service, you must delete the associated service before deleting the model version. A deleted model cannot be recovered.

Click the number in **Versions** to view the version list.

Figure 2-45 Version list



The version list displays the following information.

Table 2-27 Version list

Parameter	Description
Version	Current version of a model.
Status	Model status.
Deployment Type	Types of the services that a model can be deployed as.
Model Size	Model size.
Model Source	Model source.
Created	Model creation time.
Description	Model description.
Operation	• **Deploy**: Deploy a model as real-time services, edge services, or batch services. • **Delete**: Delete a version of a model.

Viewing Model Details

After a model is created, you can view the model information on the model details page.

1. Log in to the **ModelArts console**. In the navigation pane on the left, choose **Model Management**.
2. Click the name of the target model to access its details page.

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

   Table 2-28 Basic model information

Parameter	Description
Name	Model name.
Status	Model status.
Version	Current version of a model.
ID	Model ID.
Description	Click the edit button to add the description of a model.
Deployment Type	Types of the services that a model can be deployed as.
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.

Parameter	Description
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 model 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 for meta models from a container image.
Dynamic loading	Dynamic loading status if the meta model comes from a training job or OBS.
Size	Model size.

Parameter	Description
Health Check	Displays health check status if the meta model comes from OBS or a container image. When health check is enabled, the probe parameter settings are displayed. • **Startup Probe**: This probe checks if the application instance has started. If a startup probe is provided, all other probes are disabled until it succeeds. If the startup probe fails, the instance is restarted. If no startup probe is provided, the default status is **Success**. • **Readiness Probe**: This probe verifies whether the application instance is ready to handle traffic. If the readiness probe fails (meaning the instance is not ready), the instance is taken out of the service load balancing pool. Traffic will not be routed to the instance until the probe succeeds. • **Liveness Probe**: This probe monitors the application health status. If the liveness probe fails (indicating the application is unhealthy), the instance is automatically restarted. The probe parameters include **Check Mode**, **Health Check URL** (displayed when **Check Mode** is set to **HTTP request**), **Health Check Command** (displayed when **Check Mode** is set to **Command**), **Health Check Period**, **Delay**, **Timeout**, and **Maximum Failures**.
Model Description	Description document added during the creation of a model.
Instruction Set Architecture	System architecture.
Inference Accelerator	Type of inference accelerator cards.

Table 2-29 Model details tabs

Parameter	Description
Model Precision	Model recall, precision, accuracy, and F1 score of a model.
Parameter Configuration	API configuration, input parameters, and output parameters of a model.
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.

Parameter	Description
Events	The progress of key operations during model creation. Events are stored for three months and will be automatically cleared then. For details about how to view events of a model, see **Viewing ModelArts Model Events**.
Constraint	Displays the constraints of service deployment, such as the request mode, boot command, and model encryption, based on the settings during model creation. For models 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 a model was deployed. Click a service name to go to the service details page.

2.6.2 Viewing ModelArts Model Events

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

The following table lists the events, based on which you can locate faults occurred during model creation. The following table lists the available events.

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	The model starts to import.	N/A
Abnormal	Failed to create the image.	Locate and rectify the fault based on the error information. **FAQs**
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. **FAQs**

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.	N/A
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. **FAQs**
Abnormal	User *xxx* does not have OBS permission obs:object:PutObjectAcl.	The IAM user does not have the obs:object:PutObjectAcl permission on OBS. Add the agency permission for the IAM user. **FAQs**
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. **FAQs**
Normal	Model description updated.	N/A

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	Model runtime dependencies not updated.	N/A
Normal	Model runtime dependencies updated. Recreating the image.	N/A
Abnormal	SWR traffic control triggered. Try again later.	SWR traffic control triggered. Try again later.
Normal	The system is being upgraded. Try again later.	N/A
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.	N/A
Normal	Source image shared.	N/A
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.	N/A
Normal	Model file imported.	N/A
Normal	Model size calculated.	N/A
Abnormal	Failed to import the model.	For details, see **FAQs**
Abnormal	Failed to copy the model file. Check whether you have the OBS permission.	Check whether you have the OBS permission. **FAQs**
Abnormal	Failed to schedule the image creation task.	Contact technical support.
Abnormal	Failed to start the image creation task.	Contact technical support.

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	The Roman image has been created but cannot be shared with resource tenants.	Contact technical support.
Normal	Image created.	N/A
Normal	The image creation task started.	N/A
Normal	The environment image creation task started.	N/A
Normal	The request for creating an environment image received.	N/A
Normal	The request for creating an image received.	N/A
Normal	An existing environment image is used.	N/A
Abnormal	Failed to create the image. For details, see image creation logs.	View the build logs to locate and rectify the fault. **FAQs**
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. **FAQs**
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.
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.

Type	Event (*xxx* should be replaced with the actual value.)	Solution
Warning	Failed to calculate the model size. More than one *xxx* model file is stored in path *xxx*.	N/A

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

Viewing Events

1. In the navigation pane of the **ModelArts console**, choose **Model Deployment**. In the model list, click the target model name to access its details page.
2. View the events in the **Events** tab.

2.6.3 Managing ModelArts Model Versions

For model lineage and tuning, ModelArts provides model versioning.

Prerequisites

You have created a model in ModelArts.

Creating a Version

On the **Model Management** page, click **Create Version** in the **Operation** column of the target model. On the **Create Version** page, configure parameters. For details, see **Creating a Model**. Click **Create now**.

Deleting a Version

On the **Model Management** page, click the number in **Versions**. Click **Delete** in the **Operation** column of the target version and enter **DELETE**.

NOTE

If a model version has been deployed as a service, you must delete the associated service before deleting the model version. A deleted version cannot be recovered.

Deleting a Model

On the **Model Management** page, click **Delete** in the **Operation** column of the target model, enter **DELETE** in the text box, and click **OK** to delete the model.

NOTE

If a model version has been deployed as a service, you must delete the associated service before deleting the model version. A deleted model cannot be recovered.

2.7 Managing a Synchronous Real-Time Service

2.7.1 Viewing Details About a Real-Time Service

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

1. Log in to the **ModelArts console** and choose **Model Deployment** > **Real-Time Services**.
2. Click the name of the target service to access its details page.

   View service information. For details, see **Table 2-30**.

Table 2-30 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 you can click the edit button to modify.
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.

Parameter	Description
Runtime Log Output	This feature is disabled by default. The runtime logs of real-time services are stored only in the ModelArts log system. If this feature 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 LTS log management, see **Log Tank Service**. **NOTE** • This cannot be disabled once it is enabled. • You will be billed for log query and storage features provided by LTS. For details, see **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 2-31**.

Table 2-31 Details of a real-time service

Parameter	Description
Usage Guides	This page displays the API URL, model 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 Real-Time Service Prediction**.
Instance	Instance information of the asynchronous real-time service. The number of instances is the same as the number of instances set during service deployment. If the service is modified or the service is abnormal, the number of instances changes. To rebuild an abnormal instance, click **Delete**. After the instance is deleted, a new instance with the same compute specifications is automatically created.

Parameter	Description
Configuration Updates	This page displays **Current Configurations** and **Update History**. • **Current Configurations**: model name, version, status, instance flavor, traffic ratio, number of instances, deployment timeout interval, environment variables, and storage mounting. If the service is deployed in a dedicated resource pool, the resource pool information is also displayed. • **Update History**: historical model information.
Monitoring	This page displays resource usage and model calls. • **Resource Usage**: includes the used and available CPU, memory, GPU, and NPU resources. • **AI Application Calls**: indicates the number of model calls. The statistics collection starts after the model status changes to **Ready**. (This parameter is not displayed for WebSocket services.)
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 Events of a Real-Time Service**.

Parameter	Description
Logs	This page displays the log information about each model 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 feature 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 feature, you can combine keywords with **&&** or **||**. For example, **query logs&&erro*** or **query logs||erro***. After enabling this feature, 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 **Using TMS Tags to Manage Resources by Group**
Cloud Shell	You can use Cloud Shell provided by the ModelArts console to log in to the instance container of a running real-time service. For details, see **Using Cloud Shell to Debug a Real-Time Service Instance Container**.

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 2-32 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 2-33 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 used to call an inference request
PROJECT_ID	Project ID used to call an inference request
USER_NAME	Username used to call an inference request
USER_ID	User ID 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 == 'zhangsan'
  ```

- Example 2:

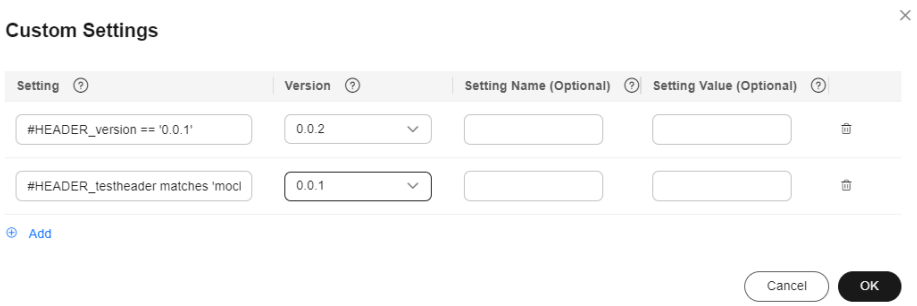
  If the account name in the inference request starts with **op**, the specified version is matched.

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

Table 2-34 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 2-46 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_**, indicating 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.

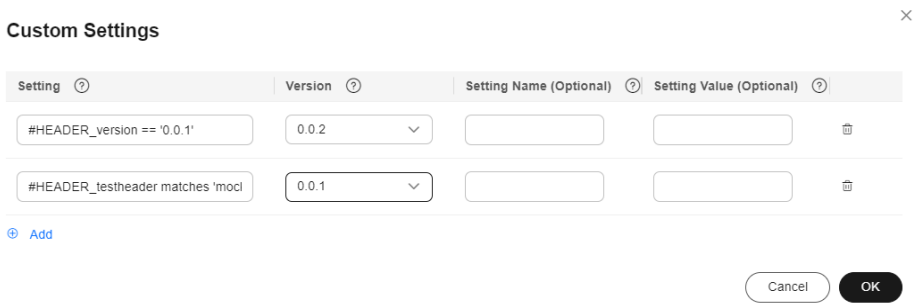
    ```
    #HEADER_testheader matches 'mock.*'
    ```

  – 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 2-47 Using the header to access different versions

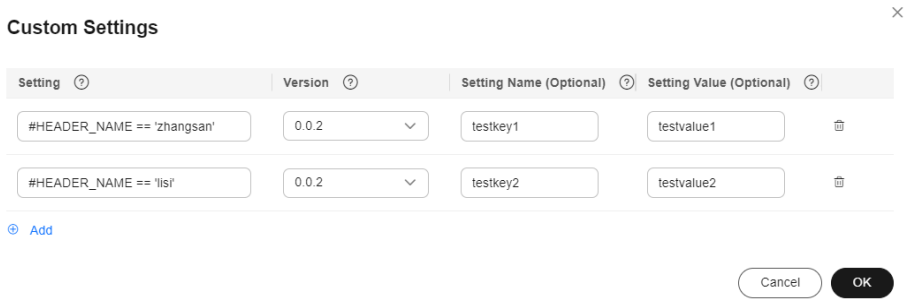
    

- If a real-time service version supports different runtime configurations, you can use **Setting Name** and **Setting Value** to specify customized runtime parameters so that different users can use different running configurations.

  Example:

  When user A accesses the model, the user uses configuration A. When user B accesses the model, 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 2-48 Customized running parameters added for a customized configuration rule



2.7.2 Viewing Events of a Real-Time Service

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 service deployment and running process and accurately locate faults when a task exception occurs. You can check the following events:

Table 2-35 Events

Event Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	The service starts to deploy.	N/A
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*	For details, see **FAQs**
Normal	The image starts to create.	N/A
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 is created.	N/A
Abnormal	Service *xxx* failed. Error: *xxx*	Locate and rectify the fault based on the error information.

Event Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	Failed to update the service. Perform a rollback.	Contact technical support.
Normal	The service is being updated.	N/A
Normal	The service is being started.	N/A
Normal	The service is being stopped.	N/A
Normal	The service has been stopped.	N/A
Normal	Auto stop has been disabled.	N/A
Normal	Auto stop has been enabled. The service will stop after *x*s.	N/A
Normal	The service stops when the auto stop time expires.	N/A
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*.	N/A
Normal	Service instances stopped in resource pool *xxx*.	N/A
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.	N/A
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.	N/A
Normal	Service *xxx* started.	N/A

Event Type	Event (*xxx* should be replaced with the actual value.)	Solution
Abnormal	Failed to start service *xxx*.	For details, 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.	N/A
Abnormal	Failed to update the service. The rollback failed.	Contact technical support.
Normal	[model 0.0.1] OBS bucket, OBS parallel file system, or SFS Turbo mounted successfully. [%s] %s volume successfully.	N/A

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

Viewing Events

1. In the navigation pane of the ModelArts console, choose Model Deployment**Service Deployment** > **Real-Time Services**. In the service list, click the target service name or ID to go to the service details page.
2. View the events in the **Events** tab.

2.7.3 Managing the Lifecycle of a Real-Time Service

Starting a Service

A service not in the **Deploying** state can be started. A service is billed from the time when it is running. To start a service, use either of the following methods:

- Log in to the ModelArts console. In the navigation pane, choose **Model Deployment** and go to the target service management page. Click **Start** in the **Operation** column to start the target service.
- Log in to the ModelArts console. In the navigation pane, choose **Model Deployment** and go to the target service management page. Click the name of the target service to access its details page. 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 console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**. Choose **More** > **Stop** in the **Operation** column to stop the target service.
- Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**. Click the name of the target service to access its details page. 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 console**. In the navigation pane, choose **Model Deployment** > **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.

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 console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**. Choose **More** > **Restart** in the **Operation** column to restart the target service.
- Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Real-Time Services**. Click the name of the target service to access its details page. Click **Restart** in the upper right corner of the page to restart the service.

NOTE

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

2.7.4 Modifying a Real-Time Service

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

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

Method 1: Modify the Service Information on the Service Management Page

Method 2: Modify the 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 services during the upgrade.
- ModelArts supports hitless rolling upgrade of real-time services in some scenarios. Prepare for and fully verify the upgrade.

Table 2-36 Scenarios for hitless rolling upgrade

Meta Model Source	Using a Public Resource Pool	Using a Dedicated Resource Pool
Training job	Not supported	Not supported
Container image	Not supported	Supported. The custom image for creating a model must meet **Specifications for Custom Images**.
OBS	Not supported	Not supported

Method 1: Modify the Service Information on the Service Management Page

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** and go to the target service management page.
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 a Model 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.

When modifying parameters of a real-time service, you can add a custom environment variable parameter to trigger service restart. For example, if a service is originally deployed in a public resource pool, the service may be scheduled to the new public resource pool after the environment variables are modified.

Method 2: Modify the Service Information on the Service Details Page

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** and go to the target service management page.
2. Click the name of the target service to access its details page.
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 a Model 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.

   When modifying parameters of a real-time service, you can add a custom environment variable parameter to trigger service restart. For example, if a service is originally deployed in a public resource pool, the service may be scheduled to the new public resource pool after the environment variables are modified.

2.7.5 Viewing Performance Metrics of a Real-Time Service on Cloud Eye

ModelArts Metrics

The cloud service platform provides Cloud Eye to help you better understand the statuses of ModelArts real-time services and model loads. 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.

Table 2-37 ModelArts metrics

ID	Name	Description	Value Range	Unit	Number System	Monitored Object	Monitoring Interval
cpu_usage	CPU Usage	CPU usage of ModelArts Unit: %	≥ 0%	%	N/A	ModelArts model loads	1 minute
mem_usage	Memory Usage	Memory usage of ModelArts Unit: %	≥ 0%	%	N/A	ModelArts model loads	1 minute
gpu_util	GPU Usage	GPU usage of ModelArts Unit: %	≥ 0%	%	N/A	ModelArts model loads	1 minute
gpu_mem_usage	GPU Memory Usage	GPU memory usage of ModelArts Unit: %	≥ 0%	%	N/A	ModelArts model loads	1 minute
npu_util	NPU Usage	NPU usage of ModelArts Unit: %	≥ 0%	%	N/A	ModelArts model loads	1 minute
npu_mem_usage	NPU Memory Usage	NPU memory usage of ModelArts Unit: %	≥ 0%	%	N/A	ModelArts model loads	1 minute
successfully_called_times	Number of Successful Calls	Times that ModelArts services have been successfully called Unit: counts/minute	≥ counts/minute	times/minute	N/A	ModelArts model loads ModelArts real-time services	1 minute

ID	Name	Description	Value Range	Unit	Number System	Monitored Object	Monitoring Interval
failed_called_times	Number of Failed Calls	Times that ModelArts services failed to be called Unit: counts/minute	≥ counts/minute	times/minute	N/A	ModelArts model loads ModelArts real-time services	1 minute
total_called_times	Total Calls	Times that ModelArts services are called Unit: counts/minute	≥ counts/minute	times/minute	N/A	ModelArts model loads ModelArts real-time services	1 minute
disk_read_rate	Disk Read Rate	Disk read rate of ModelArts Unit: bits/minute	≥ bits/minute	bits/min	N/A	ModelArts model loads	1 minute
disk_write_rate	Disk Write Rate	Disk write rate of ModelArts Unit: bits/minute	≥ bits/minute	bits/min	N/A	ModelArts model loads	1 minute
send_bytes_rate	Uplink rate	Outbound network traffic rate of ModelArts. Unit: bits/minute	≥ bits/minute	bits/min	N/A	ModelArts model loads	1 minute
recv_bytes_rate	Downlink rate	Inbound network traffic rate of ModelArts.	≥ bits/minute	bits/min	N/A	ModelArts model loads	1 minute

ID	Name	Description	Value Range	Unit	Number System	Monitored Object	Monitoring Interval
req_count_2xx	2xx Responses	Number of times that the API returns a 2xx response	≥ counts/minute	times/minute	N/A	ModelArts real-time services	1 minute
req_count_4xx	4xx Errors	Number of times that the API returns a 4xx error	≥ counts/minute	times/minute	N/A	ModelArts real-time services	1 minute
req_count_5xx	5xx Errors	Number of times that the API returns a 5xx error	≥ counts/minute	times/minute	N/A	ModelArts real-time services	1 minute
avg_latency	Average Latency	Average latency of the API	≥ ms	ms	N/A	ModelArts real-time services	1 minute
tp_99	TP99	Collects the response durations of each call over the last minute, arranges them in ascending order, and then excludes the top 1% of values. The highest remaining value represents the TP99.	≥ ms	ms	N/A	ModelArts real-time services	1 minute

ID	Name	Description	Value Range	Unit	Number System	Monitored Object	Monitoring Interval
tp_999	TP99.9	Collects the response durations of each call over the last minute, arranges them in ascending order, and then excludes the top 0.1% of values. The highest remaining value represents the TP99.9.	≥ ms	ms	N/A	ModelArts real-time services	1 minute

If a monitored object has multiple dimensions, all dimensions are mandatory when you use APIs to query the 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"
  }
  ]

Table 2-38 Dimension description

Key	Value
service_id	Real-time service ID
model_id	Model ID

Setting Alarm Rules

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.

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

Step 1 Log in to the management console.

Step 2 In the **Service List**, click **Cloud Eye** under **Management & Governance**.

Step 3 In the navigation pane on the left, choose **Alarm Management** > **Alarm Rules** and click **Create Alarm Rule**.

Step 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**.

----End

Method 2: Setting an Alarm Rule for a Single Service

Step 1 Log in to the management console.

Step 2 In the **Service List**, click **Cloud Eye** under **Management & Governance**.

Step 3 In the navigation pane, choose **Cloud Service Monitoring > ModelArts**.

Step 4 Locate a real-time service for which you want to create an alarm rule and click **Create Alarm Rule** in the **Operation** column.

Step 5 On the **Create Alarm Rule** page, create an alarm rule for ModelArts real-time services and models as prompted.

----End

Method 3: Setting an Alarm Rule for a Model Version

Step 1 Log in to the management console.

Step 2 In the **Service List**, click **Cloud Eye** under **Management & Governance**.

Step 3 In the navigation pane, choose **Cloud Service Monitoring > ModelArts**.

Step 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.

Step 5 On the **Create Alarm Rule** page, create an alarm rule for model loads as prompted.

----End

Method 4: Setting an Alarm Rule for a Metric of a Service or Model Version

Step 1 Log in to the management console.

Step 2 In the **Service List**, click **Cloud Eye** under **Management & Governance**.

Step 3 In the navigation pane, choose **Cloud Service Monitoring > ModelArts**.

Step 4 Click the down arrow next to the target real-time service name. Then, click the target version and view alarm rule details.

Step 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.

----End

Viewing Monitoring Metrics

Cloud Eye on the cloud service platform monitors the statuses of ModelArts real-time services and model loads. You can obtain the monitoring metrics of each ModelArts real-time service and model on the management console. It takes a period of time to transmit and display monitored data. The statuses displayed on the Cloud Eye console are obtained 5 to 10 minutes before. You can view the monitoring 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.

The monitoring data is unavailable without alarm rules configured in Cloud Eye. For details, see **Setting Alarm Rules**.

Step 1 Log in to the management console.

Step 2 In the **Service List**, click **Cloud Eye** under **Management & Governance**.

Step 3 In the navigation pane, choose **Cloud Service Monitoring** > **ModelArts**.

Step 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.

Step 5 In the monitoring area, you can select a period to view the monitoring data.

You can view the monitoring data in the last 1 hour, 3 hours, or 12 hours. To view the monitoring curve of a longer time range, click to enlarge the graph.

----End

2.7.6 Integrating a Real-Time Service API into the Production Environment

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.

2.7.7 Configuring Auto Restart upon a Real-Time Service Fault

Description

When an Snt9b hardware fault is detected, the system automatically resets the Snt9B chip and restarts the real-time inference service. This helps to improve the recovery speed.

Constraints

Only synchronous real-time services using Snt9b resources are supported.

This function will reset the entire node. Before enabling it, ensure that the deployed real-time service is using specifications with 8 x N cards and properly evaluate the impacts on services running on the node.

Enabling Auto Restart

When deploying a real-time service, select **Configure Now** next to **Advanced Configuration** and enable auto restart.

2.8 Managing Batch Inference Jobs

2.8.1 Viewing Details About a Batch Service

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

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Batch Services**.
2. Click the name of the target service to access its details page.

   View service information. For details, see **Table 2-39**.

Table 2-39 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.
Instances	Number of nodes of the batch service.
Start Time	Start time of the batch service job.

Parameter	Description
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.
Model Name & Version	Name and version of the model used by the batch service.
Advanced Log Management	This feature is disabled by default. The runtime logs of batch services are stored only in the ModelArts log system. If this feature 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 LTS log management, see **Log Tank Service**. **NOTE** • This cannot be disabled once it is enabled. • You will be billed for log query and storage features provided by LTS. For details, see **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 2-40**.

Table 2-40 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 Events of a Real-Time Service**.

Parameter	Description
Logs	This page displays the log information about each model 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 feature 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 feature, you can combine keywords with **&&** or **||**. For example, **query logs&&erro*** or **query logs||erro***. After enabling this feature, you can combine keywords with **AND** or **OR**. For example, **query logs AND erro*** or **query logs OR erro***.

2.8.2 Viewing Events of a Batch Service

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 service deployment and running process and accurately locate faults when a task exception occurs. You can check the following events:

Table 2-41 Events

Event Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	The service starts to deploy.	N/A
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*	For details, see **FAQs**

Event Type	Event (*xxx* should be replaced with the actual value.)	Solution
Normal	The image starts to create.	N/A
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 is created.	N/A
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.	N/A
Normal	The service is being started.	N/A
Normal	The service is being stopped.	N/A
Normal	The service has been stopped.	N/A
Normal	Auto stop has been disabled.	N/A
Normal	Auto stop has been enabled. The service will stop after *x*s.	N/A
Normal	The service stops when the auto stop time expires.	N/A
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*.	N/A
Normal	Service instances stopped in resource pool *xxx*.	N/A

Event Type	Event (*xxx* should be replaced with the actual value.)	Solution
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.	N/A
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.	N/A
Normal	Service *xxx* started.	N/A
Abnormal	Failed to start service *xxx*.	For details, 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.	N/A
Abnormal	Failed to update the service. The rollback failed.	Contact technical support.
Normal	[model 0.0.1] OBS bucket, OBS parallel file system, or SFS Turbo mounted successfully. [%s] %s volume successfully.	N/A

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

Viewing Events

1. In the navigation pane of the **ModelArts console**, choose Model Deployment**Service Deployment** > **Batch Services**. In the service list, click the target service name or ID to go to the service details page.
2. View the events in the **Events** tab.

2.8.3 Managing the Lifecycle of a Batch Service

Starting a Service

A service not in the **Deploying** state can be started. A service is billed from the time when it is running. To start a service, use either of the following methods:

- Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** and go to the target service management page. Click **Start** in the **Operation** column to start the target service.
- Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** and go to the target service management page. Click the name of the target service to access its details page. 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 console**. In the navigation pane, choose **Model Deployment** > **Batch Services**. Click **Stop** in the **Operation** column to stop the target service.
- Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** > **Batch Services**. Click the name of the target service to access its details page. 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 console**. In the navigation pane, choose **Model Deployment** > **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.

Restarting a Service

Batch services cannot be restarted.

2.8.4 Modifying a Batch Service

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

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

Method 1: Modify the Service Information on the Service Management Page

Method 2: Modify the 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 services during the upgrade.
- ModelArts supports hitless rolling upgrade of real-time services in some scenarios. Prepare for and fully verify the upgrade.

Table 2-42 Scenarios for hitless rolling upgrade

Meta Model Source	Using a Public Resource Pool	Using a Dedicated Resource Pool
Training job	Not supported	Not supported
Container image	Not supported	Supported. The custom image for creating a model must meet **Specifications for Custom Image**.
OBS	Not supported	Not supported

Method 1: Modify the Service Information on the Service Management Page

1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** and go to the target service management page.
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 batch service parameters, see **Deploying a Model as a Batch Inference Service**.

Method 2: Modify the Service Information on the Service Details Page

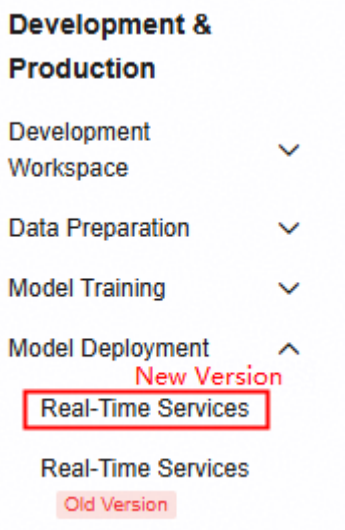
1. Log in to the **ModelArts console**. In the navigation pane, choose **Model Deployment** and go to the target service management page.
2. Click the name of the target service to access its details page.
3. Click **Modify** in the upper right corner of the page, modify the service details, and submit 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 batch service parameters, see **Deploying a Model as a Batch Inference Service**.

3 Migrating from Old-Version Real-Time Services to New-Version Real-Time Services

Scenarios

ModelArts real-time services (old-version) will soon be discontinued. For AI model deployment scenarios, users currently using the old version may experience service interruptions due to its discontinuation. The new version of ModelArts real-time services supports all large model formats and distributed deployment, providing more comprehensive metric monitoring and a smoother O&M experience. Consequently, smoothly migrating old-version services to the new version has become a common user requirement. This section describes how to create corresponding inference services on the new **Real-Time Services** page based on existing configurations and model image paths.



Overview

The new-version real-time services feature an evolved architecture, introducing KubeInfer, a brand-new distributed inference solution. By leveraging multi-node parallel computing, distributed inference significantly enhances the efficiency of large-scale models. Through architecture compatibility design, the new version

supports distributed inference while maintaining full compatibility with single-node inference scenarios. Therefore, you can deploy models and images from the old-version services directly on the new **Real-Time Services** page.

Permissions

You must have the permission set for the new inference version. For details, see **Configuring Basic Permissions for Inference Deployment**.

Billing

New-version real-time services currently can only be deployed in dedicated resource pools. Billing will be incurred for compute resources in these pools, as detailed in **Table 3-1**.

When you create a dedicated resource pool, the billing starts after the node is created. The driver upgrade and installation process is also considered as a part of the creation success.

Table 3-1 Billing items

Billing Item		Description	Billing Mode	Billing Formula
Compute resource	Dedicated resource pools	Usage of compute resources. For details, see **ModelArts Pricing Details**.	Pay-per-use	Specification unit price x Number of compute nodes x Usage duration
			Yearly/ Monthly	Specification unit price x Number of compute nodes x Purchase duration
Storage resource	EVS	Container data disk mounted by the user. If an EVS disk is mounted when you create a dedicated resource pool, the storage will be billed.	Same as the billing mode of dedicated resource pool	Specification unit price x Disk size x Number of disks x Usage duration or purchase duration

Preparations

Prepare a dedicated resource pool whose job type is model deployment. You can purchase a dedicated resource pool or change the job type of an existing dedicated resource pool.

- Purchase a dedicated resource pool: Select **Model Deployment** as the job type. For details, see **Creating a Dedicated Resource Pool**.
- Modify job types of a dedicated resource pool: Add the **Model Deployment** option to an existing pool's job types. For details, see **Modifying the Job Types Supported by a Dedicated Resource Pool**.

If your old-version inference services are still running, do not deselect the **Model Deploy (Old Version)** job type. You may deselect it only after your service migration is complete.

Step 1: Obtaining the Deployment Information of Old-Version Real-Time Services

1. Log in to the **ModelArts console** and choose **Model Deployment** > **Real-Time Services (Old Version)**.
2. Click the name of the service to be migrated. The service details page is displayed.
3. In the **Configuration Updates** tab, click the model name in the model list to view the model details and obtain the configuration information about the model image.

Figure 3-1 Model details

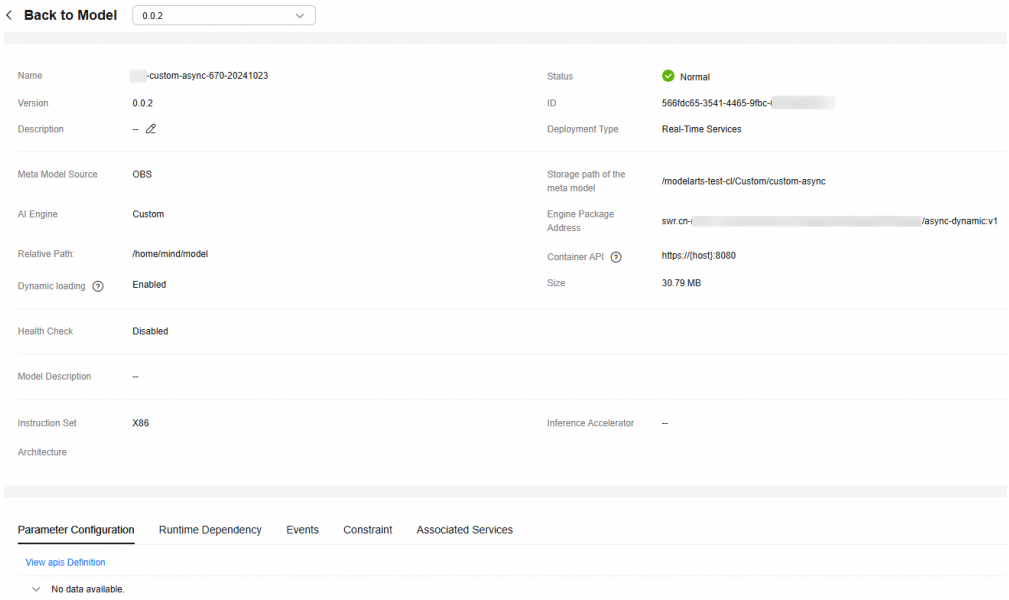


Table 3-2 describes the mapping of model and deployment parameters between the new and old versions of real-time services.

Table 3-2 Parameter mapping between the new and old versions of real-time services

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
Meta Model Source	Training job	Deployment Configuration > Model Settings > Model Source	• **Model Source**: Select **Custom Model**. • **Storage Type**: Select **Object Storage Service – Bucket**. • **Storage Location**: Enter the output path of the training job. Go to the training job details page on the console to obtain the output path. For details, see **Viewing Training Jobs and Details**. • **Mount Path**: Set it based on the new service. • **Local Storage Acceleration**: Enable this function based on new business needs.
	OBS		• **Model Source**: Select **Custom Model**. • **Storage Type**: Select **Object Storage Service – Bucket**. • **Storage Location**: Set it to the same path as the storage path of the meta model for the old real-time service. • **Mount Path**: Set it based on the new service. • **Local Storage Acceleration**: Configure this parameter based on new business needs.

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
	Container image	Deployment Configuration > Unit Settings > Image Type	No model configuration required; configure directly in the image settings. • **Image Type**: Select **Custom Images**. • **Image**: Same as the **Container Image Storage Path** in the old version.
AI Engine	TensorFlow, PyTorch, MindSpore	Deployment Configuration > Unit Settings > Image Type	• **Image Type**: Select **Preset Images**. • **Image**: Select a preset image of the required framework.
	Custom		• **Image Type**: Select **Custom Images**. • **Image**: Same as the **Container Image Storage Path** in the old version.
Container API (custom AI engine)	Protocol and port number.	Deployment Configuration > Deployment Management Settings > Container Port	Same as the old version.
Health Check (custom AI engine)	Startup, Readiness, and Liveness probes.	Deployment Configuration > Unit Settings > Health Check	Same as the old version.
Model Description	Document name and URL.	Deployment Configuration > Basic Information > Description (Optional)	Use the service description to include model details.
Instruction Set Architecture	System architecture of the model.	/	Not maintained; you must ensure the correctness of the runtime architecture.

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
Inference Accelerator	Enable/Disable accelerator cards.	/	Not maintained; you must ensure the correctness of the accelerator cards.
Deployment Type	Real-time service	/	Simply create the service on the **Model Deployment** > **Real-Time Services** page.
Boot Command	Custom boot command.	Deployment Configuration > Unit Settings > More Settings > Boot Command	Same as the old version.

4. Return to the **Real-Time Services (Old Version)** list page, click **Modify** in the **Operation** column for the service you want to migrate, and enter the modification page to obtain its deployment information.

   Table 3-3 describes the mapping of model and deployment parameters between the new and old versions of real-time services.

   Table 3-3 Parameter mapping between the new and old versions of real-time services

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
Auto Stop	Enable or disable the switch.	Service Information > Basic Information > Auto Stop	Select or deselect.
Resource Pool	**Public**/ **Dedicated Resource Pool**.	Deployment Configuration > Resource Settings > Resource Pool	The new version currently only supports dedicated resource pools. Select the pool prepared in **Preparations**.

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
Multi-Pool Load Balancing	Enable or disable the switch.	/	After creating the service, you can add multiple deployments across different pools. For details, see **Configuring Deployment Settings**.
Model and Configuration > Model Source	**My Subscriptions**.	/	Not supported.
Model and Configuration > Model Source	**My Model** & **Model and Version**.	Deployment Configuration > Model Settings > Model Source Deployment Configuration > Unit Settings > Image Type	Set this parameter based on **Meta Model Source** in **Table 3-2**.
Model and Configuration > Traffic Ratio (%)	Custom value.	After creating a service, add multiple deployments and configure different traffic percentages for them. For details, see **Traffic Policies**.	Same as the old version.
Model and Configuration > Instance Flavor	Select an instance flavor from the drop-down list.	Deployment Configuration > Unit Settings > Unit Instance Specifications	Same as the old version.
Model and Configuration > Node Affinity Deployment	Enable this function and choose the preferred nodes for deployment.	Deployment Configuration > Unit Settings > More Settings > Affinity Scheduling	Set affinity type and strength; select target nodes. Same as the old version.

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
Model and Configuration > Instances	Custom value.	Deployment Configuration > Resource Settings > Deployment Replicas	Same as the old version.
Model and Configuration > Environment Variable	Input key-value pairs.	Deployment Configuration > Unit Settings > Environment Variables	Same as the old version.
Model and Configuration > Timeout	Custom value.	Deployment Configuration > Deployment Management Settings> More Settings > Deployment Timeout (Minutes)	Same as the old version.
Model and Configuration > Mount Storage	• Storage volume type: OBS bucket/OBS parallel file system • Source address • Mount path	Storing model or code files: Deployment Configuration > Unit Settings > Mount File Storage	• **Storage Type**: Matches **Volume Type** of the old version. • **Storage Location**: Matches the source address of the old version. • **Mount Path**: Matches **Mount Path** of the old version.

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
	• Storage volume type: SFS Turbo • Source address • Mount path		• **Storage Type**: Matches **Volume Type** of the old version. • **File System**: Matches **File System Name** of the old version. • **Mount Path**: Matches **Mount Path** of the old version.
Traffic Limit	Requests per second.	Service Information > High Availability Settings > Rate Limiting Policy > Requests Per Second	Same as the old version.
WebSocket	Enable or disable the switch.	Service Information > Network Settings > Service Protocol	Select **WSS** in the new version if this function is enabled in the old version.
Advanced Log Management	Enable or disable the switch.	Service Information > Advanced Settings > Ingest Logs to LTS	Same as the old version.
Log Dump	Enable or disable the switch.	Service Information > Advanced Settings > Ingest Logs to LTS	Same as the old version.
Application Authentication	Enable or disable the switch.	Service Information > Network Settings > Authentication Mode	Not supported. You can select API key authentication in the service information.
Notification	Enable or disable the switch.	/	Not supported.

Old-Version Real-Time Service Parameter	Parameter Value	New-Version Real-Time Service Parameter	Parameter Value
Advanced Configuration > Tags	Enter tag keys and values.	Service Information > Advanced Settings > Tags	Add tags. The tag keys and values must be the same as those of the old-version service.
Advanced Configuration > IPv6 Support	Enable or disable the switch.	/	Not supported.

Step 2: Checking the Image

If **rank_table** information is used for service orchestration in the service image of the old inference version, the information is generated by the Modelarts-Infers-Operator component in the new inference version.

Table 3-4 Image **rank_table** description

Field	Description
status	**completed** indicates ready; **incomplete** indicates not ready (waiting required).
server_group_count	Number of roles under an instance.
group_id	Role ID (starting from 0). Role 0 usually provides external services.
server_count	Number of pods under the role.
server_id	Host IP address of the host machine where the pod is located.
server_ip	IP address for accessing the pod.
pod_name	Unique pod name.
device	List of NPUs used by the pod.
device_id	ID of the NPU on the host machine (not necessarily starting from 0 or consecutive).
device_ip	IP address for accessing the NPU
device_logical_id	Logical ID of the NPU in the pod (starting from 0).

Field	Description
rank_id	Logical ID of the NPU in the role (starting from 0).

```
{
   "version": "1.0",
   "status": "completed",
   "server_group_count": "2",
   "server_group_list": [
      {
         "group_id": "0",
         "server_count": "1",
         "server_list": [
            {
               "server_id": "192.xxx.xxx.238",
               "server_ip": "172.xxx.xxx.139",
               "pod_name": "infer-xxx-role-0-xxx",
               "device": [
                  {"device_id": "0", "device_ip": "29.xxx.xxx.197", "rank_id": "0"},
                  ...
               ]
            }
         ]
      },
      {
         "group_id": "1",
         "server_count": "2",
         "server_list": [
            {
               "server_id": "192.xxx.xxx.0",
               "server_ip": "172.xxx.xxx.54",
               "pod_name": "infer-xxx-role-1-xxx",
               "device": [
                  {"device_id": "0", "device_ip": "29.xxx.xxx.74", "rank_id": "0"},
                  ...
               ]
            },
            ...
         ]
      }
   ]
}
```

When using the rank table, wait until the status is **completed**. Reference script **rank_table_checker.py**:

```
import json
import os
import time

GLOBAL_RANK_TABLE_ENV = 'GLOBAL_RANK_TABLE_FILE_PATH'
POD_IP_ENV = 'POD_IP'

def wait_completed_global_rank_table():
   while True:
      try:
         pod_ip = os.getenv(POD_IP_ENV)
         global_rank_table_path = os.getenv(GLOBAL_RANK_TABLE_ENV)
         if not global_rank_table_path:
            print('read env \"{}\" failed'.format(GLOBAL_RANK_TABLE_ENV))
         with open(global_rank_table_path, 'r') as file:
            buf = file.read()
         rank_table = json.loads(buf)
         if rank_table["status"] == "completed":
```



```
            server_group_list = rank_table['server_group_list']
            for group in server_group_list:
                server_list = group["server_list"]
                for i in range(len(server_list)):
                    if server_list[i]["server_ip"] == pod_ip:
                        return
            print("cannot find local ip in ranktable!")
        else:
            print("status of ranktable is not completed!")
    except Exception as e:
        print(e)
    time.sleep(1)

if __name__ == "__main__":
    wait_completed_global_rank_table()
```

Step 3: Creating a New-Version Real-Time Service

1. Log in to the **ModelArts console** and choose **Model Deployment** > **Real-Time Services**.
2. In the real-time service list, click **Deploy**.
3. On the **Deploy Real-Time Service** page, configure the service and deployment information of the new-version real-time service based on the old-version settings in **Table 3-2** and **Table 3-3**. For details about more parameters, see *User Guide* > "Deploying a Model as a Real-Time Service."
4. On the **Deploy Real-Time Service** > **Confirmation** page, confirm the configuration information and click **Confirm Deployment**.

   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.

   You can go to the real-time service list to check if the deployment is complete. Once the service status changes from **Deploying** to **Running**, the service is deployed.

Follow-Up Operations

After the deployment is complete, you can verify the service on the prediction page. For details, see **Real-Time Service Prediction**.

If the service works correctly, follow these guides to set up public network access: **Accessing a Real-Time Service via Private Network through Dedicated APIG, WAF, VPC, and ELB** and **Accessing a Real-Time Services via Private Network through Load Balancing**.

