ModelArts

Model Call

Issue 01

Date 2026-07-17

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 Quickly Calling Preset Models

ModelArts integrates mainstream third-party models, including the Qwen series, and provides OpenAI-compatible APIs and end-to-end model services. These models can be deployed on ModelArts with just few clicks. This section describes the preset models supported by ModelArts and how to deploy and call a model with just few clicks.

Introduction to Preset Models

ModelArts provides open-source models. On the **ModelArts console**, choose **Asset Management** > **Models** > **Preset Models** to view the supported models, model types, supported operations, supported resource types, and number of accelerators.

Click the target model card. On the model details page, view the model details, such as the basic information, supported capabilities, and inference features. You can select a proper model for training and inference based on the information and integrate it into your enterprise solution.

NOTE

The models available in each region may vary.

Figure 1-1 Preset Models

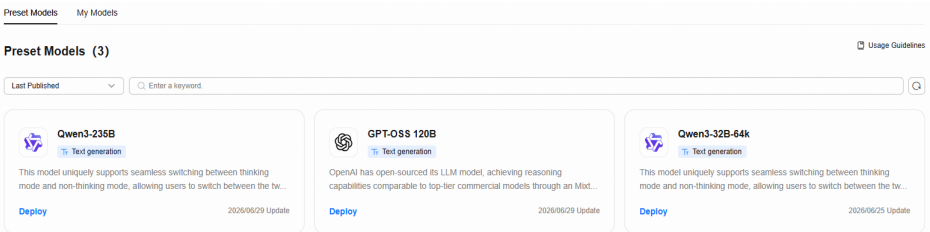
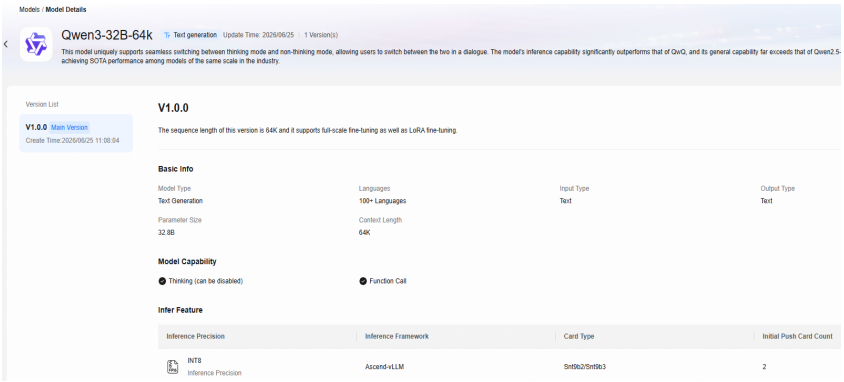


Figure 1-2 Model details page



Supported Models

On the **ModelArts console**, choose **Asset Management** > **Models** > **Preset Models**. On the model card, click **Deploy** to deploy the target model. For details, see **Calling a Preset Model**. For details about model capabilities, see **Introduction to Preset Models**.

NOTE

The models available in each region may vary.

Table 1-1 Supported models

Series	Model Name	Model Type	Version	model Parameter	Request URL	Call Method
Qwen	Qwen3-235B	Text generation	V1.0.0	qwen3-235b	/v1/chat/completions	OpenAI-compatible Chat API
	Qwen3-32B-64k	Text generation	V1.0.0	qwen3_32b	/v1/chat/completions	OpenAI-compatible Chat API
GPT-OSS	GPT-OSS 120B	Text generation	V1.0.0	gpt-oss-120b	/v1/chat/completions	OpenAI-compatible Chat API

Calling a Preset Model

After deploying a model on ModelArts, you can perform inference by calling the model via its API. Once you have obtained an API key, you can use cURL or code to trigger model requests. Procedure:

1. **Deploy a model with one click.**
   a. Log in to the **ModelArts console**, choose **Asset Management** > **Models** > **Preset Models**, and click **Deploy** on the model card.
   b. On the **Create Service** panel, configure related information and click **OK**.

Table 1-2 Parameters for creating a service

Parameter	Description	Example Value
Model	Name of the model you want to deploy.	Qwen3-32B-64k
Service Name	Name used to identify and manage the real-time service. Enter a name as prompted. The value can contain 1 to 64 characters, including letters, digits, hyphens (-), and underscores (_).	**service-test**
Resource Pool Type	Public resource pools and dedicated resource pool are supported. • **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. To select a dedicated resource pool, create one in advance.	**Public resource pool**

Parameter	Description	Example Value
Inference Unit	Select the hardware resource configuration for the real-time service instances.	Use the recommended value.
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 this option. Default: **1 hour**.

2. **Obtain an API Key**

   a. On the **ModelArts console**, choose **Model Inference** > **Real-Time Inference** and click **API Key Authorization Management**.

   b. In the upper right corner, click **Create API Key**. Enter the API key name and description, select the authorization scope as required, 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.

   - If you select **Specified real-time services**, go to the **next step** to bind the API key.
   - If you select **All real-time services**, skip the **next step**.

   c. (Optional) In the **API Key Authorization Management** tab, click **Bind** in the **Operation** column of the API key, select the model service you want to bind to the API key, and click **OK**.

3. **Obtain the API URL**

   This guide uses a shared gateway by default. After deployment, the API URL and token information is available on the service details page.

   – API URL: On the **Model Inference** > **Real-Time Inference** page, click your deployed service to enter the details page. In **Network Settings**, copy the **Public API URL**.

   – Model request URL: For the Qwen3-32B-64k model, the request URL is **/v1/chat/completions**. For more model request URLs, see the **Request URL** column in the **preceding model list**.

4. **Call the Model via cURL or Python**

   You can perform inference by modifying the sample code below. Simply update the full API URL, model parameter, and API key:

   – Inference API URL: Combine the public API URL and the model request URL (obtained in **3**).

- API key: Use the key obtained in **2**.
- **model** parameter: Use the **model** parameter listed in the **preceding table**. The following uses Qwen3-32B-64k as an example. You can modify the code as needed.

Python

```
import requests
import json

if __name__ == '__main__':
    url = "https://***/v2/infer/***/v1/chat/completions" # Real-time service URL = Public API URL + Model request URL
    api_key = "API_KEY" # Replace API_KEY with the obtained API key.

    # Send request.
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {api_key}'
    }
    data = {
        "model": "qwen3_32b", # Model
        "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Hello"}
        ]
    }
    response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)

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

Curl

```
curl -X POST "https://***/v2/infer/***/v1/chat/completions" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $API_KEY" \
  -d '{
    "model": "qwen3_32b",
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Hello"}
    ]
  }'
```


2 Text Generation

In the process of developing AI applications, developers often need to handle tasks related to text understanding and generation. However, traditional text processing methods are typically inefficient and yield suboptimal results. LLMs possess capabilities for text understanding and text-based dialogue. For example, when you input text information, the LLM can understand the content and generate appropriate responses based on that information. Nevertheless, efficiently leveraging these LLMs remains a challenge. This tutorial will guide you on how to use the model service API to call the model for text understanding and content generation. You can also build or extend your own applications or automate tasks using this API, thereby enhancing development efficiency and application quality.

Application Scenarios

You can use the model's text generation capabilities in the following scenarios.

Table 2-1 Application scenarios

Scenario	Use Case	Description
Content creation	Article generation	Automatically generates practical texts such as articles, news, and comments to improve content production efficiency.
	Text polishing	Aids authors in creative brainstorming and text refinement for news reports and blog posts.
Intelligent interaction	Intelligent customer service	Generates natural and smooth responses in the customer service system to enhance user experience.
	Chatbot	In fields such as online consultation and English learning, understand user intent, generate responses according to requirements.

Scenario	Use Case	Description
Personalized teaching	Subject question answering	Analyzes the question, explain the key points, outline the solution approach, and present the results.
	Language learning	As required, engages in conversations in certain languages to help users become accustomed to daily communication in the target language.
Machine translation	Automatic translation	Leverages speech models to achieve simultaneous interpretation, daily subtitle generation, and text language translation.
Work processing	Data processing	Processes the data and tasks as required, such as reading research reports, analyzing news, and evaluating content.

Billing

Inference service deployment is billed based on duration. Costs are incurred when the status is **Running** or **Alarm**. Stop the service when not in use. For details, see **Inference Deployment Billing Items**.

Prerequisites

- The model service to be called has been deployed on ModelArts.
- You have obtained the service URL, API key, and **model** parameter. For details, see **Calling a Preset Model**.

API Reference

For the complete parameters for model calls, see **Model API Call Guide**.

Writing Prompts

Prompt is the text input by users when interacting with LLMs to guide the model in generating the desired response. Properly designed and written prompts can enhance the quality and accuracy of the model's output. The core functions of prompts:

- Define the task: Instruct the model on what to do (for example: translation, summarization, creation, inference, etc.).
- Provide context: Add background information to make the response more relevant and tailored to the request.
- Constraint output format: Specify the structure of the response (such as list, JSON, code, etc.).

- Control style and tone: Adjust the professionalism, conciseness, or creativity of the response.

In the Chat API, the messages object is used to pass prompt information to the model. The **role** field defines the role of the message sender, while the **content** field carries the message content. The model interprets the content based on the provided role and generates an appropriate response.

- **User Message**

  The end user sends messages to the model, at which point the **role** field should be set to **user**. This type of message typically contains specific tasks or information that the user wants the model to process.

  The following is a simple user message requesting the model to introduce itself.

```
"messages": [
    { "role": "user", "content": "Introduce yourself. "}
]
```

- **System Message**

  Used to set the model's long-term personality, behavioral guidelines, and conversation context. In this case, the **role** field should be set to **system**. If you configure system messages, place them in the first position of the **messages** list.

  The following is an example of a system message that indicates the model is an assistant and restricts the content of its responses.

```
"messages": [
    {"role": "system", "content": " You are a helpful and concise assistant. Keep your responses to no more than two sentences." }
]
```

- **Model Message**

  Used to provide conversation history, in which case the **role** field should be set to **assistant**. In multi-turn conversations, you need to pass in the conversation history, so that the model can remember what it said before and maintain coherence.

```
"messages": [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello"},
    {"role": "assistant", "content": "Hello! Nice to meet you! I am your AI assistant and I'm here to help you with anything you need. Whether it's answering questions, assisting with analysis, or just having a chat, I'm ready to assist. Is there anything specific you need help with?"},
    {"role": "user","content": "Introduce yourself" }
]
```

Single-Turn Dialogue

Interact with the model in a single turn of dialogue, where the model returns content based on system and user messages.

Since it is non-streaming output, the model needs to process all content before returning it to you in one go, which may introduce some latency.

The following is an example code for a single-turn conversation. You can replace the model by modifying the **model** parameter. For details about the **model** parameter, see **Creating a Chat Request**.

Curl

```
curl -X POST "https://***/v2/infer/***/v1/chat/completions" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $API_KEY" \
-d '{
    "model": "qwen3_32b",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello"}
    ]
}'
```

Python

```
import requests
import json
if __name__ == '__main__':
    url = "https://***/v2/infer/***/v1/chat/completions"  # API address
    api_key = "API_KEY"  # Replace API_KEY with the obtained API key.
    # Send request.
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {api_key}'
    }
    data = {
         "model": "qwen3_32b",  # Model
         "messages": [
            {
                "role": "system",
                "content": "You are a helpful assistant."
            },
            {
                "role": "user",
                "content": "Hello"
            }
        ]
     }
    response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)
    # Print result.
    print(response.status_code)
    print(response.text)
```

Multi-Turn Dialogue

Combining system messages, model messages, and user messages enables multi-turn conversations, allowing for multiple dialogues on a single topic.

Note: chat.completions API is stateless. In each request, all historical information is included in the **messages** field and the **role** field is set to inform the model of previous conversations from different roles, ensuring contextually relevant and continuous dialogue.

The following is an example code for a multi-turn conversation. You can replace the model by modifying the **model** parameter. For details about the **model** parameter, see **Creating a Chat Request**.

Curl

```
curl -X POST "https://***/v2/infer/***/v1/chat/completions" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $API_KEY" \
-d '{
    "model": "qwen3_32b",
    "messages": [
```



```
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello"},
        {"role": "assistant", "content": "Hello! Nice to meet you!"},
        {"role": "user", "content": "Introduce yourself."}
    ]
}'
```

Python SDK

```
import requests
import json
if __name__ == '__main__':
    url = "https://***/v2/infer/***/v1/chat/completions"  # API address
    api_key = "API_KEY"  # Replace API_KEY with the obtained API key.
    # Send request.
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {api_key}'
    }
    data = {
         "model": "qwen3_32b",  # Model
         "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Hello"}
            {"role": "assistant", "content": "Hello! Nice to meet you!"},
            {"role": "user", "content": "Introduce yourself."}
        ]
     }
    response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)
    # Print result.
    print(response.status_code)
    print(response.text)
```

Streaming Output

With the LLM output, dynamic content is displayed as it is generated. This allows users to see intermediate outputs without waiting for the entire inference process to complete, improving the user experience by reducing perceived wait times (viewing content as it is produced).

The following is an example of stream output code. You can replace the model by modifying the **model** parameter. For details about the **model** parameter, see **Creating a Chat Request**.

Curl

```
curl -X POST "https://***/v2/infer/***/v1/chat/completions" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $API_KEY" \
-d '{
    "model": "qwen3_32b",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello"}
    ],
    "stream": true
}'
```

Python SDK

```
import requests
import json
if __name__ == '__main__':
    url = "https://***/v2/infer/***/v1/chat/completions"  # API address
```



```
api_key = "API_KEY"  # Replace API_KEY with the obtained API key.
# Send request.
headers = {
    'Content-Type': 'application/json',
    'Authorization': f'Bearer {api_key}'
}
data = {
     "model": "qwen3_32b",  # Model
     "messages": [
         {"role": "system", "content": "You are a helpful assistant." },
         {"role": "user","content": "Hello" }
     ],
     "stream": True
 }
response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)
# Print result.
print(response.status_code)
print(response.text)
```

Function Call

Function calling enables models to dynamically call external APIs by defining tool interface specifications, facilitating seamless function expansion.

The following is an example code using function calling. You can replace the model by modifying the **model** parameter. For details about the **model** parameter, see **Creating a Chat Request**.

Python SDK example:

```
import requests
import json
def get_weather(location: str, unit: str):
    return f"Getting the weather for {location} in {unit}..."

if __name__ == '__main__':
    url = "https://***/v2/infer/***/v1/chat/completions"  # API address
    api_key = "API_KEY"  # Replace API_KEY with the obtained API key.

    # Send request.
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {api_key}'
    }
    data = {
        "model": "qwen3_32b",  # Model
        "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Beijing weather"}
        ],
        "tools": [
            {
                "type": "function",
                "function": {
                    "name": "get_weather",
                    "description": "Get weather information for a specified location",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "location": {
                                "type": "string",
                                "description": "City and state, e.g., 'San Francisco, CA'"
                            }
                        },
                        "required": [
                            "location"
                        ]
```



```
                }
            }
        }
    ],
    "thinking": {
        "type": "enabled"  # Specifies whether to enable deep thinking. It is disabled by default.
    }
}
response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)
# Print result.
print(response.status_code)
print(response.text)
```

Setting the Model Response Length Limit

To control costs or adjust response length, for example, limiting a response to 500 characters, you can configure the **max_tokens** field in your request to set a maximum output length.

The following is an example code for setting the model response length limit. You can replace the **model** parameter with your desired model. For details about the **model** parameter, see **Creating a Chat Request**.

Curl

```
curl -X POST "https://***/v2/infer/***/v1/chat/completions" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $API_KEY" \
-d '{
    "model": "qwen3_32b",
    "messages": [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Hello"}
    ],
    "max_tokens": 1024
}'
```

Python SDK

```
import requests
import json
if __name__ == '__main__':
    url = "https://***/v2/infer/***/v1/chat/completions"  # API address
    api_key = "API_KEY"  # Replace API_KEY with the obtained API key.
    # Send request.
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {api_key}'
    }
    data = {
         "model": "qwen3_32b",  # Model
         "messages": [
             {"role": "system", "content": "You are a helpful assistant." },
             {"role": "user", "content": "Hello"}
         ],
         "max_tokens": 1024
     }
    response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)
    # Print result.
    print(response.status_code)
    print(response.text)
```


3 Model API Call Guide

ModelArts provides powerful real-time inference. You can deploy models on dedicated instances. This chapter describes the specifications for calling chat APIs.

API Information

Table 3-1 API information

Parameter	Description	Example Value
API URL	API URL for calling the model service.	• API URL: On the **Model Inference** > **Real-Time Inference** page, click your deployed service to enter the details page. Copy the **Public API URL**.
model	**model** parameter in an API call	For details, see the **model** parameter in **Quickly Calling Preset Models**.

Chain of Thought (CoT)

CoT refers to the model's ability to generate a series of intermediate reasoning steps when solving complex problems. This capability allows the model not only to provide the final answer but also to demonstrate its reasoning process, thereby enhancing the model's explainability and transparency.

Creating a Chat Request

- Authentication description

  Inference services support API key authentication. The authentication header is in the following format:

  ```
  'Authorization': 'Bearer API key of the region where the service is deployed'
  ```

- The request and response parameters are as follows.

Table 3-2 Request parameters

Parameter	Mandatory	Default Value	Type	Description
model	Yes	None	Str	Model to call. For details about the value, see the **model** parameter in **Quickly Calling Preset Models**.
messages	Yes	-	Array	Input question. **role** shows the role, and **content** shows the dialog content. Example: "messages": [{"role": "system","content": "You are a helpful AI assistant."}, {"role": "user","content": "Which number is larger, 9.11 or 9.8?"}] For more information, see **Table 3-3**.
messages.prefix	No	false	Boolean	Controls whether to enable continuation mode. In this mode, the user provides a message starting with **assistant**, and the model completes the rest based on that beginning and the input instruction. To use this feature, ensure that the last message in the **messages** list has the **role** set to **assistant** and the **prefix** parameter set to **true**. Example: messages = [{"role": "user", "content": "Write a snippet of Python code"}, {"role": "assistant", "content": "```python\n", "prefix": True}]
stream_options	No	None	Object	Specifies whether to display the number of used tokens during streaming output. This parameter is only valid when **stream** is set to **True**. You need to set **stream_options** to **{"include_usage": true}** to print the number of tokens used. For more information, see **Table 3-4**.

Parameter	Mandatory	Default Value	Type	Description
max_tokens	No	None	Int	Maximum number of tokens that can be generated for the current task, including tokens generated by the model and reasoning tokens for deep thinking.
top_k	No	-1	Int	The candidate set size determines the sampling range during generation. For example, setting it to **50** means only the top 50 scoring tokens are sampled at each step. A larger size increases randomness; a smaller one makes the output more predictable.
top_p	No	1.0	Float	Nucleus sampling. It keeps only the words with combined probabilities above the threshold p and removes the rest. These selected words are then normalized and sampled again. Lower settings reduce word options, making outputs focused and cautious. Higher settings expand word choices, creating varied and creative outputs. Adjust either **temperature** or **top_p** separately for best results, not both at once. Value range: 0 to 1. The value **1** indicates that all tokens are considered.
temperature	No	1.0	Float	Model sampling temperature. The higher the value, the more random the model output; the lower the value, the more deterministic the output. Adjust either **temperature** or **top_p** separately for best results, not both at once.

Parameter	Mandatory	Default Value	Type	Description
stop	No	None	None/ Str/ List	A list of strings used to stop generation. The output does not contain the stop strings. For example, if the value is set to ["You," "Good"], text generation will stop once either **You** or **Good** is reached.
stream	No	False	Bool	Controls whether to enable streaming inference. The default value is **False**, indicating that streaming inference is disabled.
n	No	1	Int	Number of responses generated for each input message. • If **beam_search** is not used, the recommended value range of n is 1 ≤ n ≤10. If n is greater than 1, ensure that **greedy_sample** is not used for sampling, that is, **top_k** is greater than 1 and **temperature** is greater than 0. • If **beam_search** is used, the recommended value range of n is 1 < n ≤ 10. If n is 1, the inference request will fail. **NOTE** For optimal performance, keep n at 10 or below. Large values of n can significantly slow down processing. Inadequate video RAM may cause inference requests to fail.

Parameter	Mandatory	Default Value	Type	Description
use_beam_search	No	False	Bool	Controls whether to use **beam_search** to replace sampling. When this parameter is used, the following parameters must be configured as required: • **n**: > **1** • **top_p**: **1.0** • **top_k**: **-1** • **temperature**: **0.0**
presence_penalty	No	0.0	Float	Applies rewards or penalties based on the presence of new words in the generated text. The value range is [-2.0,2.0].
frequency_penalty	No	0.0	Float	Applies rewards or penalties based on the frequency of each word in the generated text. The value range is [-2.0,2.0].
length_penalty	No	1.0	Float	Imposes a larger penalty on longer sequences in a beam search process. When this parameter is used, the following parameters must be configured as required: • **top_k**: **-1** • **use_beam_search**: **true** • **best_of**: > **1**

Parameter	Mandatory	Default Value	Type	Description

chat_template_kwargs.thinking	No	false	Bool	Specifies whether to enable the CoT. For some models, you can enable the CoT by adding the template parameter **"thinking": true** when initiating an inference request.
				Example of enabling the CoT:
				{
				"model": "**DeepSeek-V31**",
				"messages": [{
				"role": "system",
				"content": "You are a helpful assistant."
				}, {
				"role": "user",
				"content": "Hello"
				}],
				"chat_template_kwargs": {
				"thinking": true
				}
				}

chat_template_kwargs.enable_thinking	No	true	Bool	Specifies whether to enable the CoT. For some models, you can disable the CoT by adding the template parameter **"enable_thinking": false** when initiating an inference request.
				Example of disabling the CoT:
				{
				"model": "qwen3-32b",
				"messages": [{
				"role": "system",
				"content": "You are a helpful assistant."
				}, {
				"role": "user",
				"content": "Hello"
				}],
				"chat_template_kwargs": {
				"enable_thinking": false
				}
				}

Table 3-3 Request parameter **messages**

Parameter	Mandatory	Default Value	Type	Description
role	Yes	None	Str	Different roles correspond to different message types. • **system**: developer-entered instructions like response formats and roles for the model to follow. • **user**: user-entered messages including prompts and context information. • **assistant**: responses generated by the model. • **tool**: information returned by the tool when the model calls it.

content	Yes	None	Str	• When **role** is set to **system**, this parameter indicates the AI model's personality.
				`{ "role": "system", "content": "You are a helpful AI assistant." }`
				• When **role** is set to **user**, this parameter indicates the question asked by the user.
				`{ "role": "user", "content": "Which one is larger, 9.11 or 9.8?" }`
				• When **role** is set to **assistant**, this parameter indicates the content output by the AI model.
				`{"role": "assistant","content": "9.11 is larger than 9.8."}`
				• When **role** is set to **tool**, this parameter indicates the responses returned by the tool when the model calls it.
				`{"role": "tool", "content": "The weather in Shanghai is sunny today. The temperature is 10°C."}`

Table 3-4 Request parameter **stream_options**

Parameter	Mandatory	Default Value	Type	Description
include_usage	No	true	Bool	Specifies whether the streaming response outputs token usage information. • **true**: Each chunk outputs a **usage** field that shows the total token usage. • **false**: The token usage is not displayed.

Table 3-5 Response parameters

Parameter	Type	Description
id	Str	Unique ID of the request.
object	Str	chat.completion type: Multi-turn dialogs are returned.
created	Int	Timestamp.
model	Str	Model to call.
choices	Array	Model output, including the **index** and **message** parameters. In **message**: • **content** is the model's final reply. • **reasoning content** is the model's deep thinking content (for DeepSeek models only).
usage	Object	Statistics on tokens consumed by the request: • This parameter is returned by default for non-streaming requests. • This parameter is returned by default for streaming requests. Each chunk outputs a **usage** field that shows the token usage. Parameters: • **prompt tokens**: number of input tokens. • **completion tokens**: number of output tokens. • **total tokens**: total number of tokens.

Parameter	Type	Description
prompt_logprobs	Float	Log probability. You can use this to measure the model's confidence in its output or to explore other options the model provides.

Qwen3-235B Text Generation Request Example

This section demonstrates the basic usage of a text generation model, using the Qwen3-235B model as an example to request a text response via a Python script, cURL command, or OpenAI SDK.

- Python request example:

```
import requests
import json

if __name__ == '__main__':
    url = ""https://***/v2/infer/***/v1/chat/completions""  # API address
    api_key = "API_KEY"  # Replace API_KEY with the obtained API key.

    # Send request.
    headers = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {api_key}'
    }
    data = {
        "model": "qwen3-235b,"  # Model name
        "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Hello"}
        ],
        "chat_template_kwargs": {
            "thinking": True  # Specifies whether to enable deep thinking. It is disabled by default.
        }
    }
    response = requests.post(url, headers=headers, data=json.dumps(data), verify=False)

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

- cURL request example:

```
curl -X POST "https://***/v2/infer/***/v1/chat/completions" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $API_KEY" \
  -d '{
    "model": "qwen3-235b",
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "Hello"}
    ],
     "chat_template_kwargs": {
       "thinking": true
     }
  }'
```

- OpenAI SDK request example:

```
from openai import OpenAI

base_url = "https://***/v2/infer/***/v1/chat/completions"  # API address
api_key = "API_KEY"  # Replace API_KEY with the obtained API key.

client = OpenAI(api_key=api_key, base_url=base_url)
```



```
response = client.chat.completions.create(
    model="qwen3-235b",  # Model name
    messages=[
        {"role": "system", "content": "You are a helpful assistant"},
        {"role": "user", "content": "Hello"},
    ],
    extra_body={
        "chat_template_kwargs": {
            "thinking": True  # Specifies whether to enable deep thinking. It is disabled by default.
        }
    }
)

print(response.choices[0].message.content)
```

Response Example

The following is an example response of the text generation model.

```
{
    "id":"chat-71406e38b0d248c9b284709f8435****",
    "object":"chat.completion",
    "created":1740809549,
    "model":"qwen3-235b",
    "choices":[
        {
            "index":0,
            "message":{
                "role":"assistant",
                "content":"\n\n Compare 9.11 and 9.8.:\n\n1. **Compare the integer part**: The integer part of both is 9, which is equal.\n2. **Compare the tenths place**:\n - The tenths place of 9.11 is **1**\n - 9.8 can be considered as 9.80, and its tenths place is **8**\n - **8 > 1**, so 9.8 is larger.\n\n**Conclusion**:\n**9.8 > 9.11**\n(When comparing decimals, line up the digits and compare them directly.)",
                "reasoning_content": "Well, I now need to compare 9.11 and 9.8 which is larger. First of all, I have to recall the method of comparing decimals. When comparing decimals, start by comparing the integer parts. If the integer parts are the same, compare the tenths and hundredths of the decimal parts in sequence until the larger number is determined. \n\n The integer parts of the two numbers are both 9, so they are the same. Next, compare the tenths. The tenth digit of 9.11 is 1, while the tenth digit of 9.8 is 8. This can be problematic, as some people might directly treat 9.8 as 9.80, or focus on comparing the digits in the tenths place. \n\n Now, comparing the tenths place, 9.8 has an 8, while 9.11 has a 1. Clearly, 8 is greater than 1. So, should we conclude that 9.8 is greater than 9.11? \n\n However, it is important to note that some people might incorrectly assume that the more decimal places a number has, the larger its value. But this is not true; for instance, 0.9 is greater than 0.8999. Thus, having more decimal places does not necessarily mean a larger value. \n\n Additionally, the decimal parts of the two numbers can be aligned to have the same number of digits for comparison. For instance, 9.8 can be written as 9.80, where the tenths place is 8 and the hundredths place is 0. On the other hand, for 9.11, the tenths place is 1 and the hundredths place is 1. Since 8 in the tenths place is greater than 1, 9.80 (which is 9.8) is greater than 9.11. \n\n Therefore, the final conclusion is that 9.8 is larger than 9.11.\n",
                "tool_calls":[]
                },
                "logprobs":null,
                "finish_reason":"stop",
                "stop_reason":null
        }
    ],
    "usage":{
        "prompt_tokens":21,
        "total_tokens":437,
        "completion_tokens":416
        },
    "prompt_logprobs":null
}
```


4 Model Asset Management

Introduction to Model Assets

Model asset management is the core module of ModelArts, responsible for uniformly managing both **Built-in Models** and **My Models**. This module aims to provide standardized input objects for downstream tasks including model deployment.

Built-in Models: Popular models pre-configured in ModelArts for quick and easy deployment.

My Models: Models that have been locally imported or generated after pre-training through the platform. For details about how to import a model locally, see **Managing My Models**.

Constraints

Built-in models are read-only resources and cannot be modified by users.

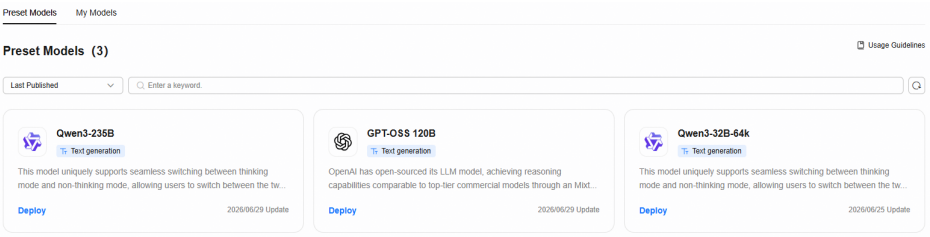
Viewing Preset Models

1. Log in to the **ModelArts console**. In the left navigation pane, choose **Asset Management** > **Models**. In the **Preset Models** tab, view the preset models. The models are displayed in card format. Each card includes the model name, model description, supported capabilities, and more.

   NOTE

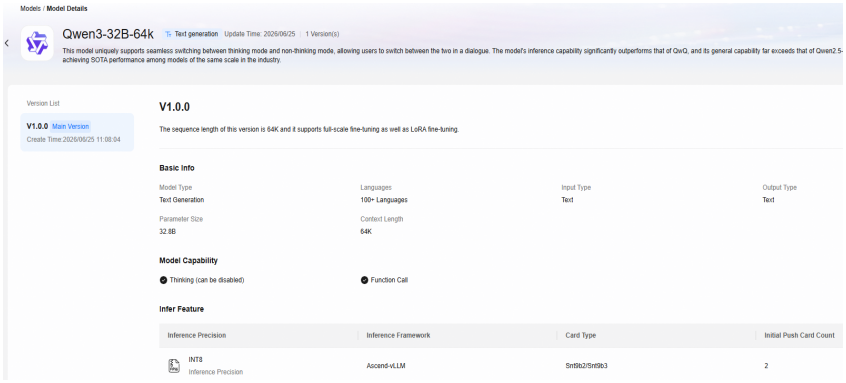
   The models available in each region may vary.

Figure 4-1 Preset models



2. Click **Deploy** on the model card as needed. Clicking the button allows you to directly access the corresponding operation.
3. Click the card to view details about the model, including **basic information, capabilities, training features, and inference features**. You can also click **Deploy** in the upper right corner.

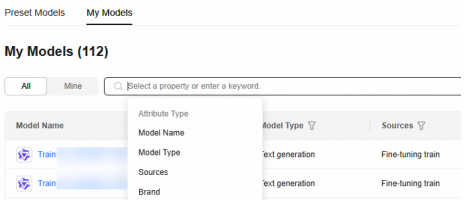
Figure 4-2 Model details



Managing My Models

1. Log in to the **ModelArts console**. On the ModelArts console, choose **Asset Management** > **Models** from the left navigation pane. Choose **My Models** to view all models in this space and the list of models created by the current user. You can filter model assets by model name, model type, source, and brand.

Figure 4-3 My models



2. Locate a model; the **Operation** column supports the following operations:
   - **Deploy**: Click to open the model deployment panel on the right and deploy the model directly.
   - **Edit Properties**: Click to modify the model asset name and description.
3. Click **Import Model** in the upper right corner of **My Models** to add open-source model assets. This function allows you to import custom models to ModelArts for further development, enhancing user model flexibility. Procedure:
   a. Click **Import Model**.
   b. Configure the basic information and version information of the model.

      Basic information:

- **Model Name**: The name of the imported open-source model. Must be 2–128 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed. Must **start with** a letter, and **end with** a letter or digit.
- **Model Type**: The modality of the imported model. Currently supported types include text generation, image understanding, image generation, video generation, and other.
- **Brand**: Supports major global open-source model platforms (e.g., DeepSeek and Qwen).
- **Model Weight File Address**: Manually enter the address or click  to select the OBS address of the model weight files. The OBS address must start with **obs://** or **/** and end with a slash (**/**). It cannot contain double slashes (**//**) except in the prefix. For example, **obs://bucketname/path/** or **/bucketname/path/**.
- **Model Description**: A brief description of the imported open-source model. This field is optional and limited to 256 characters.

Version information:

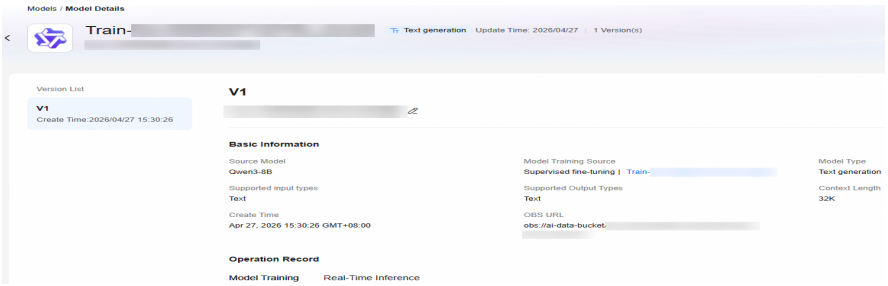
- **Model Asset Version**: For the initial import, the platform sets the version number to V1 by default.
- **Version Description**: Additional information regarding this specific version. This field is optional.

c. Click **OK**.

4. Click the name of any model in the model list to view its details. The details are as follows:
   - **Model Asset Version**: On the left side of the model details page, multiple version numbers may exist for the same model. During model training, if **New version** is selected for **Publish to Assets**, the system will generate a new version number based on the source model version selected for that training task. Consequently, a single model can contain multiple version iterations.
   - **Basic Information**: Shows basic details about the model asset, such as the source model, training task, model type, brand, input and output types, context length, creator, creation time, and OBS address.

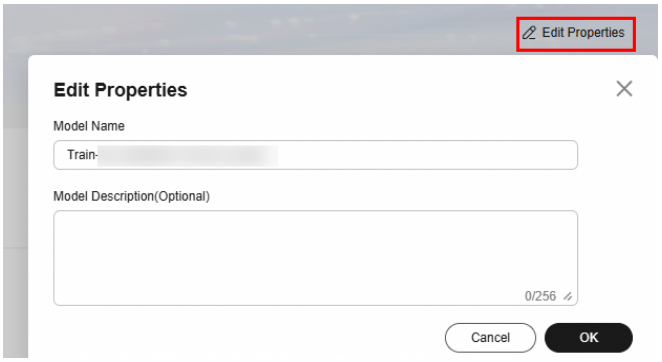
     If the model has been imported, the OBS address will display the specific path of the imported model.
   - **Operation Record**: Model assets can be used for both model training and real-time inference. The operation records section on the details page displays a comprehensive log of all tasks where this model was used for training or deployment. Under the **Associated Task Name** column, you can click a specific entry to navigate directly to that task's detail page.

Figure 4-4 Model details



5. Click the required button in the upper right corner of the model details page to perform the following operations:
   - **Edit Properties**: Click this button to modify the model name and description (optional).

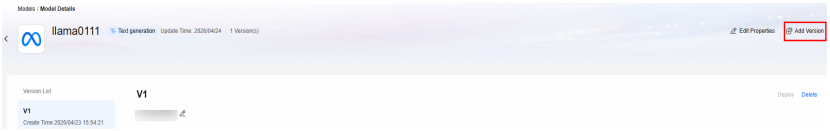
     Figure 4-5 Editing properties

   - Add a version: If the model source is an imported model asset, you can add a version. After you click **Add Version**, the model version number automatically increases by 1. Enter the address of the model weight file and model description for the new version to generate a new version for the model.

     Model Weight File Address: Manually enter the address or click to select the OBS address of the model weight files. The OBS address must start with **obs://** or **/** and end with a slash (**/**). It cannot contain double slashes (**//**) except in the prefix. For example, **obs://bucketname/path/** or **/bucketname/path/**.

     Figure 4-6 Adding a version

   - **Deploy**: Click **Deploy** to create a deployment task. For details, see **Deploying a Model as a Real-Time Service**.

- **Delete**: In the upper right corner of the model details page, click **Delete**. In the **Delete** dialog box, click **OK** to delete the model asset. If the model to be deleted is running, the model cannot be deleted.

  CAUTION

  The deletion operation is highly risky. Confirm that the model is no longer in use before proceeding with the deletion.

