ModelArts

Workflow

Issue   01
Date   2024-04-30

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Contents

1 MLOps Overview

What Is MLOps?

Machine Learning Operations (MLOps) are a set of practices with machine learning (ML) and DevOps combined. With the development of ML, it is expected not only to make breakthroughs in academic research, but also to systematically implement these technologies in various scenarios. However, there is a significant gap between academic research and the implementation of ML technologies. In academic research, an AI algorithm is developed for a certain dataset (a public dataset or a scenario-specific dataset). The algorithm is continuously iterated and optimized for this specific dataset. Scenario-oriented systematical AI development involves the development of both models and the entire system. Then, the successful experience in software system development "DevOps" is naturally introduced to AI development. However, in the AI era, traditional DevOps cannot cover the entire development process of an AI system.

DevOps

Development and Operations (DevOps) are a set of processes, approaches, and systems that facilitate communication, collaboration, and integration between software development, O&M, and quality assurance (QA) departments. DevOps is a proven approach in large-scale software system development. DevOps not only accelerates the interaction and iteration between services and development, but also resolves the conflicts between development and O&M. Development pursues speed, while O&M requires stability. This is the inherent and root conflict between development and O&M. Similar conflicts occur during the implementation of AI applications. The development of AI applications requires basic algorithm knowledge as well as fast, efficient algorithm iteration. Professional O&M personnel pursue stability, security, and reliability. Their professional knowledge is quite different from that of AI algorithm personnel. O&M personnel have to understand the design and ideas of algorithm personnel for service assurance, which are difficult for them to achieve. In this case, the algorithm personnel are required to take end-to-end responsibilities, leading to high labor cost. This method is feasible if a small number of models are used. However, when AI applications are implemented on a large scale, manpower will become a bottleneck.

MLOps Functions

The ML development process consists of project design, data engineering, model building, and model deployment. AI development is not a unidirectional pipeline job. During development, multiple iterations of experiments are performed based on the data and model results. To achieve better model results, algorithm engineers perform diverse data processing and model optimization based on the data features and labels of existing datasets. Traditional AI development ends with a one-off delivery of the final model output by iterative experimentation. As time passes after an application is released however, model drift occurs, leading to worsening effects when applying new data and features to the existing model. Iterative experimentation of MLOps forms a fixed pipeline which contains data engineering, model algorithms, and training configurations. You can use the pipeline to continuously perform iterative training on data that is being continuously generated. This ensures that the AI application of the model, built using the pipeline, is always in an optimum state.

Figure 1-1 MLOps
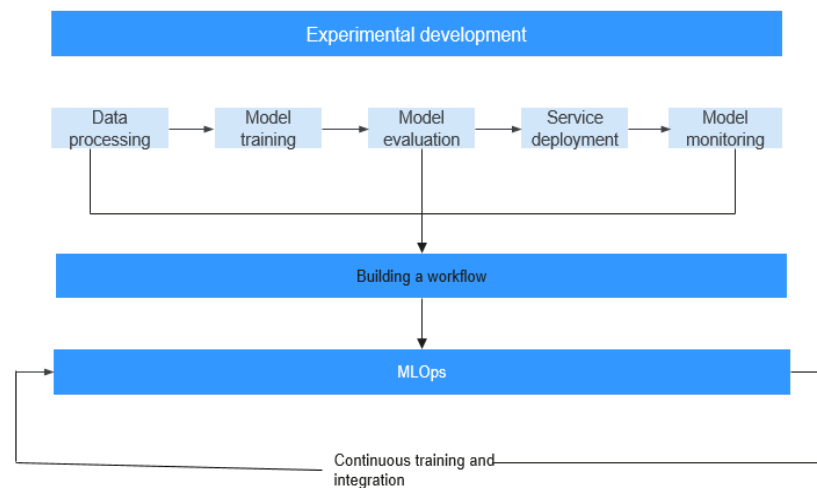
An entire MLOps link, which covers everything from algorithm development to service delivery and O&M, requires an implementation tool. Originally, the development and delivery processes were conducted separately. The models developed by algorithm engineers were delivered to downstream system engineers. In this process, algorithm engineers are highly involved, which is different from MLOps. There are general delivery cooperation rules in each enterprise. When it comes to project management, working process management needs to be added to AI projects as the system does not simply build and manage pipelines, but acts as a job management system.

The tool for the MLOps link must support the following features:

- Process analysis: Accumulated industry sample pipelines help you quickly design AI projects and processes.

- Process definition and redefinition: You can use pipelines to quickly define AI projects and design workflows for model training and release for inference.

- Resource allocation: You can use account management to allocate resource quotas and permissions to participants (including developers and O&M personnel) in the pipeline and view resource usage.

- Task arrangement: Sub-tasks can be arranged based on sub-pipelines. Additionally, notifications can be enabled for efficient management and collaboration.

- Process quality and efficiency evaluation: Pipeline execution views are provided, and checkpoints for different phases such as data evaluation, model evaluation, and performance evaluation are added so that AI project managers can easily view the quality and efficiency of the pipeline execution.

- Process optimization: In each iteration of the pipeline, you can customize core metrics and obtain affected data and causes. In this way, you can quickly determine the next iteration based on these metrics.

2 What Is Workflow?

A workflow is a pipeline tool developed based on service scenarios for deploying models or applications. In ML, a pipeline may involve data labeling, data processing, model development and training, model evaluation, application development, and application evaluation.

Figure 2-1 Workflow
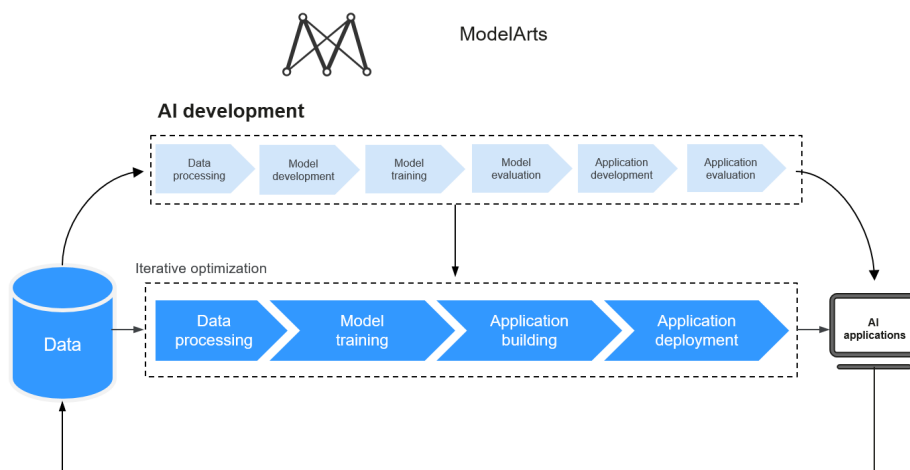
Different from traditional ML-based model building, workflows can be used to develop production pipelines. Based on MLOps, workflows enable runtime recording, monitoring, and continuous running. The development and continuous iteration of a workflow are separated in products based on roles and concepts.

A pipeline consists of multiple phases. The functions required by the pipeline and the function parameters are called through workflow SDKs. When developing a pipeline, you can use SDKs to describe phases and the relationships between phases. Developing a pipeline is the development state of the workflow. After a pipeline is determined, you can consolidate and provide it for others to use. You do not need to pay attention to what algorithms are used in the pipeline or how the pipeline is implemented. Instead, you only need to check whether the models or applications produced by the pipeline meet the release requirements. If not, you need to check whether the data and parameters need to be adjusted for iteration. Using such a consolidated pipeline is the running state of the workflow.

The development and running states of a workflow are as follows:

- Development state: Workflow Python SDKs are used to develop and debug a pipeline.
- Running state: You can configure and run a produced pipeline in visualized mode.

Leveraging DevOps principles and practices, workflows orchestrate ModelArts capabilities to help you efficiently train, develop, and deploy AI models.

Different functions are implemented in the development and running states of a workflow.

Workflow Development State

Based on service requirements, you can use Python SDKs provided by ModelArts workflows to offer each ModelArts capability as a step in a pipeline. This is a familiar and flexible development mode for AI developers. Python SDKs support:

- Debugging: partially execution, fully execution, and debugging.
- Release: Release a workflow from the development state to the running state.
- Experiment record: for persistence and the management of experiments.

Workflow Running State

Workflows are executed in visualized mode. You only need to pay attention to some simple parameter settings, whether the model needs to be retrained, and model deployment.

Running workflows are released from the development state or subscribed to from AI Gallery.

A running workflow supports:

- Unified configuration management: The parameters and resources required for a workflow are centrally managed.
- Workflow operations: include starting, stopping, copying, and deleting workflows.
- Running record: records historical running parameters and statuses of the workflow.

Workflow Components

A workflow is the description of a directed acyclic graph (DAG). You can develop a DAG through a workflow. A DAG consists of phases and the relationships between phases. To define a DAG, specify the execution content and sequence on phases. A green rectangle indicates a phase, and the link between phases shows the phase relationship. A DAG is actually an ordered job execution template.

Sample Workflows

ModelArts provides abundant scenario-oriented sample workflows. You can subscribe to them in **AI Gallery**.

Subscribing to and Using an AI Gallery Workflow

1. Log in to **AI Gallery workflow cases**.

2. On the workflow asset page of AI Gallery, select and subscribe to a workflow, read and agree to *Data Security and Privacy Risk* and *Service Agreement of AI Gallery*, and click **Continue**.

3. After the subscription, click **Run**. You will be automatically redirected to the ModelArts console. Select an asset version, workflow name, service region, and workspace, and click **Import**. The workflow details page is displayed.

4. Click **Configure** in the upper right corner. On the configuration page that appears, set parameters and click **Save** in the upper right corner to save the configuration.

5. Click **Start** in the top right corner to start the workflow.

6. On the workflow execution page, wait for the workflow to start running.

7. On the dashboard, check the status of each phase. The workflow runs automatically from one phase to the next until it finishes all the phases.

3 How to Use a Workflow?

3.1 Configuring a Workflow

3.1.1 Configuration Entries

Before or during the execution of a workflow, configure the parameters and resources required by the workflow. After obtaining the workflow, modify the configuration as required so that the produced model or application is better suited for your needs.

Workflow configurations include the configurations before and during the workflow execution.

Configurations Before Workflow Execution

Log in to the ModelArts console and choose **Workflow** to go to the workflow list page. There are two entries to configure a workflow before it runs.

- Click **Configure** in the **Operation** column of the target workflow to go to the workflow configuration page.

Figure 3-1 Configure
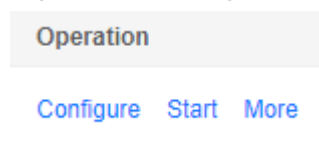
- On the workflow list page, click the name of the target workflow. On the workflow details page that is displayed, click **Configure** in the upper right corner.

Figure 3-2 Configure
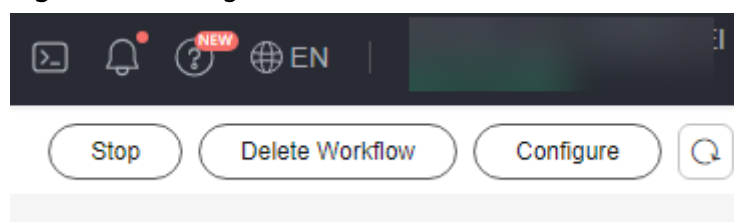
Configurations During Workflow Execution

Certain phases require parameter configuration during the execution of a workflow. When the workflow runs to such a phase, it pauses and waits for your input.

On the workflow overview page, view the to-dos on the right. Click the workflow name to go to the phase in the awaiting input status. Set the parameters for the phase and click **Next**.

Figure 3-3 Workflow to-dos
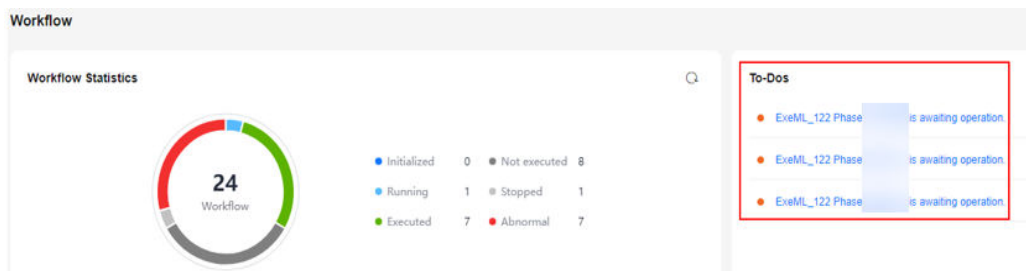
3.1.2 Runtime Configurations

Work directories can be centrally managed in ModelArts workflow. The root directory is configured in **Runtime Configurations**.

1. On the workflow list page, click the name of the target workflow.
2. Click **Configure** in the upper right corner.
3. On the **Workflow Configurations** tab page, configure the **Runtime Configurations** settings.

Figure 3-4 Runtime Configurations

3.1.3 Resource Configurations

You can configure resources for multiple phases within a workflow, with the ability to specify different configurations for each phase. The billing mode for resources consumed by a workflow is the same as that for training jobs and real-time inference, and the fees are only generated while a phase is running.

Figure 3-5 Resource Configurations
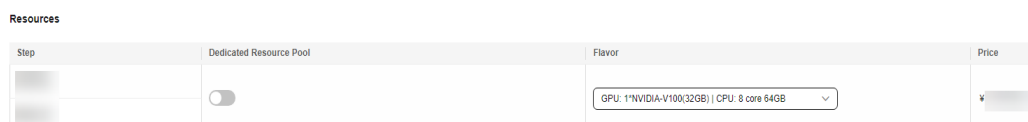
If you need to use a dedicated resource pool, enable **Dedicated Resource Pool**.

Configuring inference resource specifications

Specify the required inference resource specifications when the workflow runs on the service deployment phase.

1. Wait until the workflow runs on the service deployment phase, and the phase enters the **Awaiting input** status.
2. In the **Input** area, select the required inference resource specifications.
3. Click **Next**.

Figure 3-6 Input configurations

NOTE

- **Specifications**: The CN North-Beijing4 region supports limited-time free specifications, but each user can create only one instance using the free specifications. Other specifications are billed on a pay-per-use basis. After using the specifications, stop the workflow in a timely manner to avoid unnecessary fees.

- You can choose the package that you have bought when you select specifications. On the configuration fee tab, you can view your remaining package quota and how much you will pay for any extra usage.

3.1.4 Tab Configuration

You can filter workflows by tag for easy classification, which saves a lot of time.

Configuring Tags

1. On the ModelArts console, choose **Workflow** from the navigation pane. The workflow list page is displayed.

2. Locate the workflow you want to tag and click its name. The workflow details page is displayed.

3. Click ✎ in the upper left corner.

4. In the **Edit Workflow** dialog box that appears, enter a tag in the **Tag** text box and click **Add Tag**. The new tag is displayed below. You can add multiple tags at a time. After the tags are added, click **Yes**.

Figure 3-7 Edit button
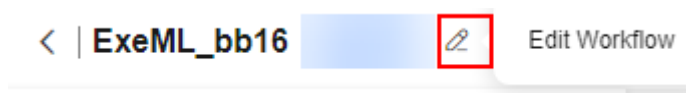
Figure 3-8 Adding a tag

Searching for a Workflow by Tag

Workflows with tags can be filtered by tag in the search box.

1. In the search box above the workflow list, set **Property** to **Tag**.

Figure 3-9 Searching for a workflow by tag
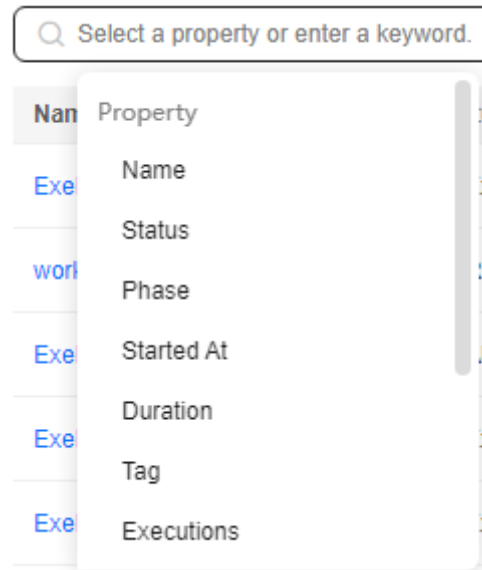
2. On the tag list that appears, click the target tag. The workflow list displays workflows with that tag.

3.1.5 SMN

Simple Message Notification (SMN) can be enabled for workflows. After you select the status to be monitored from the event list, you will be notified when an event occurs. To subscribe to notifications, enable **Subscription Notification**.

- After you enable **Subscription Notification**, specify an SMN topic. Otherwise, create an SMN topic on the SMN management console.

- You can subscribe to events for a single phase or multiple phases in a workflow, or for workflow statuses. In the subscription list, a row indicates the subscription for a phase or an entire workflow. To obtain notifications for status changes of multiple phases, add one subscription for each phase.

- You can select multiple subscription events for each subscription object, including **Awaiting input**, **Executed**, and **Abnormal**.

3.1.6 Input and Output Configurations

You can set input and output parameters on the configuration page, or when the workflow is running.

When a workflow is running, you can configure parameters for the phase in the **Awaiting input** state.

Input Configurations

The following table describes the parameters you need to specify.

Table 3-1 Input parameters

Input Parameter	Description
dataset	Select an existing dataset or create a new one.
obs	Select your OBS path.
label task	Select a labeling job under your dataset.
service	Select a deployed real-time service.
swr image	Select the image storage path required for registering the model.

Output Configurations

Click **Select** to select the OBS path to store the output data.

3.1.7 Phase Parameters

You can configure different parameters for each phase.

3.1.8 Saving Configurations

On the workflow configuration page, click **Save** in the upper right corner after you complete the configuration.

Figure 3-10 Saving Configurations

After the workflow is saved, click **Start** in the upper right corner of the page. In the dialog box that is displayed, click **OK**. The workflow is started and the runtime page is displayed.

3.2 Starting, Stopping, Searching for, Copying, or Deleting a Workflow

Starting a Workflow

When a workflow is not running, you can start it in any of the following ways:

- On the workflow list page, click **Start** in the **Operation** column. In the displayed dialog box, click **OK**.

- On the runtime configuration page, click **Start** in the upper right corner. In the displayed dialog box, click **OK**.

- On the workflow configuration page, click **Start** in the upper right corner. In the displayed dialog box, click **OK**.

  📖 **NOTE**

  > After a workflow is started, you will be charged on a pay-per-use basis. After the workflow is complete, you can stop it to avoid unnecessary fees.

Searching for a Workflow

On the workflow list page, you can use the search box to quickly search for workflows based on workflow properties.

1. Log in to the ModelArts console. In the navigation pane, choose **Workflow**.

2. In the search box above the workflow list, filter workflows based on the required property, such as the name, status, current phase, start time, running duration, or tag.

3. Click ⚙ on the right of the search box to set the content you want to display on the workflow list page and modify other display settings.

   - **Table Text Wrapping**: This function is disabled by default. If you enable this function, excess text will move down to the next line; otherwise, the text will be truncated.

   - **Operation Column**: This function is enabled by default. If you enable this function, the **Operation** column is always fixed at the rightmost position of the table.

   - **Custom Columns**: By default, all items are selected. You can select columns you want to see.

   Figure 3-11 Settings
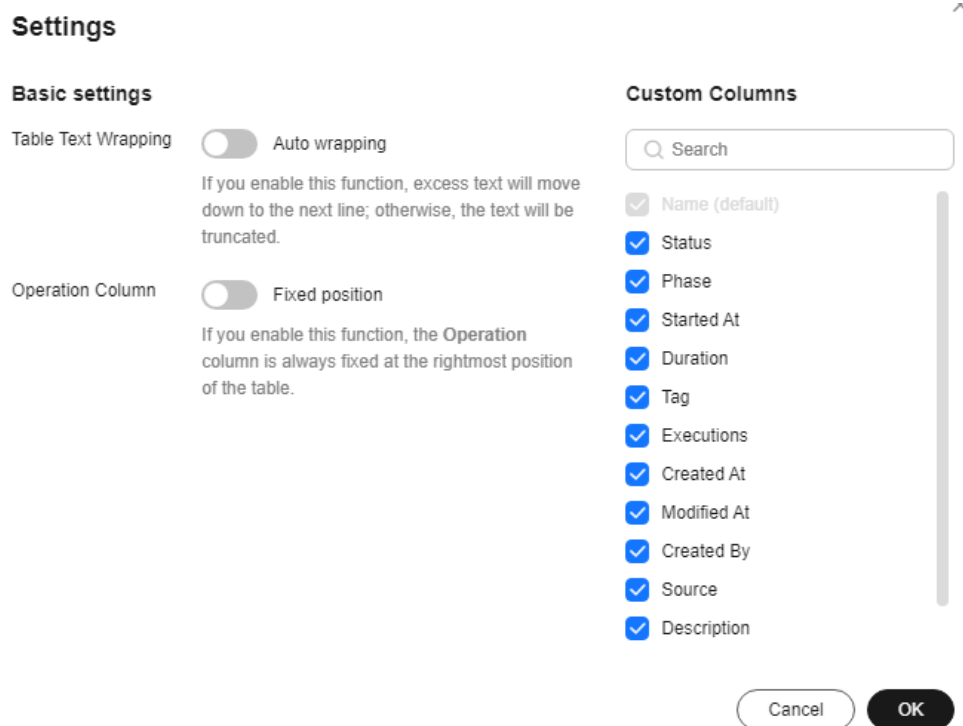
4. Click **OK**.

5. To arrange workflows by a specific property, click ⬍ in the table header.

Stopping a Workflow

You can stop a running workflow in either of the following ways:

- Workflow list page

  When a workflow is running, the **Stop** button is available in the **Operation** column. Click **Stop**. In the displayed dialog box, click **OK**.

- Click the name of a running workflow and click **Stop** in the upper right corner of the displayed page. In the displayed dialog box, click **OK**.

  📖 **NOTE**

  The **Stop** button is available only for a workflow that is running.

  After a workflow is stopped, the associated training jobs and real-time services are also stopped.

Copying a Workflow

A workflow can have only one running instance. If you want to concurrently run a workflow, copy the workflow. To do so, click **More** in the **Operation** column and select **Copy**. In the displayed dialog box, a new name is automatically generated in the format of "Original workflow name_copy".

You can rename the new workflow. Ensure that the name complies with naming specifications.

📖 **NOTE**

A workflow name is 1 to 64 characters long, starting with a letter and containing only letters, digits, underscores (_), and hyphens (-).

Deleting a Workflow

You can delete a workflow in either of the following ways:

- Workflow list page

  a. Click **More** in the **Operation** column and select **Delete**.

  b. In the displayed dialog box, enter **delete** and click **OK**.

- Runtime configuration page

  Click **Delete** in the upper right corner of the page. In the displayed dialog box, enter **DELETE** and click **OK**.

  📖 **NOTE**

  - Deleted workflows cannot be recovered.

  - After a workflow is deleted, the corresponding training jobs and real-time services are not deleted accordingly. To delete them, go to the **Training Management** > **Training Jobs** and **Service Deployment** > **Real-Time Services** pages.

3.3 Viewing Workflow Execution Records

All runtime statuses of a workflow are recorded.

1. On the workflow list page, click the name of the target workflow.
2. On the workflow details page, view all runtime records of the workflow in the left pane.

    Figure 3-12 Viewing execution records
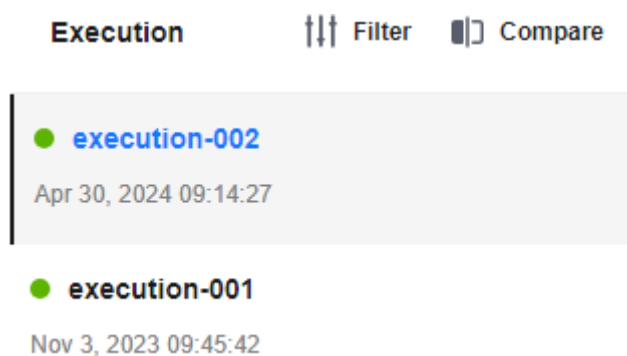
3. Delete or edit the runtime records, or rerun the workflow.
    – To delete a runtime record that is no longer needed, click **Delete**. In the displayed dialog box, click **Yes**.
    – To distinguish a runtime record from others, click **Edit Tag** to add a tag to it.
    – To rerun the workflow, click **Rerun** on a runtime record.
4. Filter and compare all runtime records of the workflow.
    – **Filter**: You can filter all runtime records by status or tag.

    Figure 3-13 Filtering
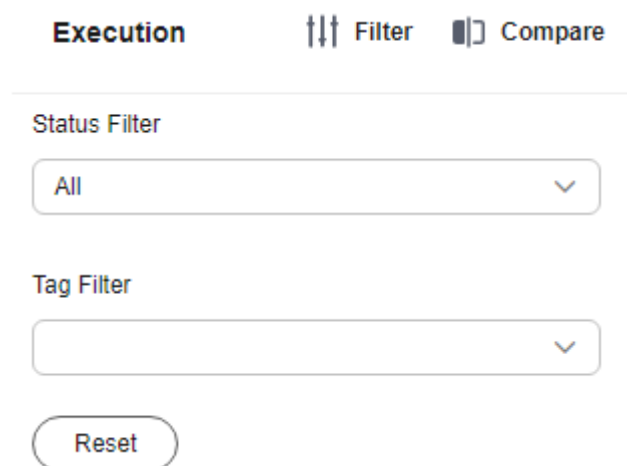
    – **Compare**: You can compare all runtime records by status, execution record, start time, duration, and metrics.

Figure 3-14 Comparison

After you click **Start** to run a workflow, the execution record list is refreshed. In addition, the data is updated on both the DAG and dashboard. An execution record is added after each startup.

You can click any phase on the workflow details page to obtain the phase status, including attributes (status, start time, and duration), input location, output location, and parameters (dataset labeling job name).

3.4 Retrying, Stopping, or Proceeding a Phase

- Retrying a phase

  If executing a single phase failed, you can click **Retry** to re-execute the current phase without restarting the workflow. Before the retry, you can modify configurations on the **Global Configuration** page. The modification takes effect after the affected phase is retried.

- Stopping a phase

  Click a phase to view its details. On this page, you can stop the running phase.

- Proceeding a phase

  If parameters need to be configured during the runtime of a single phase, the phase is awaiting operation. After the parameters are configured, you can click **Proceed** to proceed to the execution of the current phase.

3.5 Partial Execution

To reduce the time consumed by repeated execution in large-scale and complex workflows, you can choose specific phases to execute in sequence.

- Creation

  Predefine the phases to be executed when you use the SDK to create a workflow. For details, see **Partial Execution**.

- Configuration

  When configuring a workflow, enable **Execute Certain Phases**, select phases to be executed, and configure parameters for these phases.

- Start

  After saving the configuration, click **Start** to execute certain phases.

4 How to Develop a Workflow?

4.1 Concepts

4.1.1 Workflow

A workflow is a DAG that consists of phases and the relationships between phases.

A directed line segment shows the dependency between phases. The dependency decides the order of phase execution. In this example, the workflow runs from left to right after it starts. The DAG can handle the multi-branch structure as well. You can design the DAG flexibly according to the real situation. In the multi-branch situation, phases in parallel branches can run at the same time.

Table 4-1 Workflow

Parameter	Description	Mandatory	Data Type
name	Workflow name. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter.	Yes	str
desc	Workflow description	Yes	str
steps	Phases contained in a workflow	Yes	list[Step]
storages	Unified storage objects	No	Storage or list[Storage]
policy	Workflow configuration policy, which is used for partial execution	No	Policy

4.1.2 Step

A step is the smallest unit of a workflow. In a DAG, a step is also a phase. Different types of steps have different service abilities. The main parts of a step are as follows.

Table 4-2 Step

Parameter	Description	Mandatory	Data Type
name	Phase name. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter.	Yes	str
title	Title of a phase, which is displayed in the DAG. If this parameter is not configured, the name is displayed by default.	No	str
step_type	Type of a phase, which determines the function of the phase	Yes	enum
inputs	Inputs of a phase	No	AbstractInput or list[AbstractInput]
outputs	Outputs of a phase	No	AbstractOutput or list[AbstractOutput]
properties	Node properties	No	dict
policy	Phase execution policy, which includes the phase scheduling interval, the phase execution timeout interval, and the option to skip phase execution	No	StepPolicy
depend_steps	List of dependency phases. This parameter determines the DAG structure and phase execution sequence.	No	Step or list[Step]

Table 4-3 StepPolicy

Parameter	Description	Mandatory	Data Type
poll_interval_seconds	Phase scheduling interval. The default value is 1 second.	Yes	str

Parameter	Description	Mandatory	Data Type
max_execution_minutes	Phase execution timeout interval. The default value is 10080 minutes, that is, 7 days.	Yes	str
skip_conditions	Conditions that determine whether a phase is skipped	No	Condition or condition list

Step is a superclass of a phase. It has a conceptual role and is not used directly by you. Different types of phase are created based on functions, including **CreateDatasetStep**, **LabelingStep**, **DatasetImportStep**, **ReleaseDatasetStep**, **JobStep**, **ModelStep**, **ServiceStep** and **ConditionStep**. For details, see **Phase Type**.

4.1.3 Data

Data objects are used for phase input and are classified into the following types:

- Actual data objects, which are specified when you create a workflow
  - Dataset: defines existing datasets. This object is used for data labeling and model training.
  - LabelTask: defines existing labeling jobs. This object is used for data labeling and dataset version release.
  - OBSPath: defines an OBS path. This object is used for model training, dataset import, and model import.
  - ServiceData: defines an existing service. This object is used only for service update.
  - SWRImage: defines an existing SWR path. This object is used for model registration.
  - GalleryModel: defines a model subscribed from AI Gallery. This object is used for model registration.
- Placeholder data objects, which are specified when a workflow is running
  - DatasetPlaceholder: defines datasets to be specified when a workflow is running. This object is used for data labeling and model training.
  - LabelTaskPlaceholder: defines labeling jobs to be specified when a workflow is running. This object is used for data labeling and dataset version release.
  - OBSPlaceholder: defines an OBS path to be specified when a workflow is running. This object is used for model training, dataset import, and model import.
  - ServiceUpdatePlaceholder: defines existing services to be specified when a workflow is running. This object is used only for service update.
  - SWRImagePlaceholder: defines an SWR path to be specified when a workflow is running. This object is used for model registration.

- ServiceInputPlaceholder: defines model information required for service deployment when a workflow is running. This object is used only for service deployment and update.

- DataSelector: supports multiple data types. Currently, this object can be used only on the job phase (only OBS or datasets are supported).

- Data selection object:

  DataConsumptionSelector: selects a valid output from the outputs of multiple dependency phases as the data input. This object is usually used for conditional branching. (When creating a workflow, the output of which dependency phase will be used as the data input source is not specified. The data input source should be automatically selected based on the actual execution status of the dependency phases.)

Table 4-4 Dataset

Parameter	Description	Mandatory	Data Type
dataset_name	Dataset name	Yes	str
version_name	Dataset version	No	str

Example:

```
example = Dataset(dataset_name = "**", version_name = "**")
# Obtain the dataset name and version name in the ModelArts dataset module.
```

◻ NOTE

When a dataset is used as the input of a phase, configure **version_name** based on service requirements. For example, **version_name** is not required for LabelingStep and ReleaseDatasetStep, but mandatory for JobStep.

Table 4-5 LabelTask

Parameter	Description	Mandatory	Data Type
dataset_name	Dataset name	Yes	str
task_name	Labeling job name	Yes	str

Example:

```
example = LabelTask(dataset_name = "**", task_name = "**")
# Obtain the dataset name and labeling job name in the ModelArts dataset module.
```

Table 4-6 OBSPath

Parameter	Description	Mandatory	Data Type
obs_path	OBS path	Yes	str, Storage

Example:

```
example = OBSPath(obs_path = "**")
# Obtain the OBS path from Object Storage Service.
```

Table 4-7 ServiceData

Parameter	Description	Mandatory	Data Type
service_id	Service ID	Yes	str

Example:

```
example = ServiceData(service_id = "**")
# Obtain the service ID in ModelArts Real-Time Services. This object describes a specified real-time service
and is used for service update.
```

Table 4-8 SWRImage

Parameter	Description	Mandatory	Data Type
swr_path	SWR path to a container image	Yes	str

Example:

```
example = SWRImage(swr_path = "**")
# Container image path, which is used as the input for model registration
```

Table 4-9 GalleryModel

Parameter	Description	Mandatory	Data Type
subscription_id	Subscription ID of a subscribed model	Yes	str
version_num	Version number of a subscribed model	Yes	str

Example:

```
example = GalleryModel(subscription_id="**", version_num="**")
# Subscribed model object, which is used as the input of the model registration phase
```

Table 4-10 DatasetPlaceholder

Parameter	Description	Mandatory	Data Type
name	Name	Yes	str
data_type	Data type	No	DataTypeEnum
delay	Whether the data object is configured when the phase is running. The default value is **False**.	No	bool
default	Default value of a data object	No	Dataset

Example:

```
example = DatasetPlaceholder(name = "**", data_type = DataTypeEnum.IMAGE_CLASSIFICATION)
# Dataset object placeholder. Configure data_type to specify supported data types.
```

Table 4-11 OBSPlaceholder

Parameter	Description	Mandatory	Data Type
name	Name	Yes	str
object_type	OBS object type. Only "file" and "directory" are supported.	Yes	str
delay	Whether the data object is configured when the phase is running. The default value is **False**.	No	bool
default	Default value of a data object	No	OBSPath

Example:

```
example = OBSPlaceholder(name = "**", object_type = "directory" )
# OBS object placeholder. You can set object_type to file or directory.
```

Table 4-12 LabelTaskPlaceholder

Parameter	Description	Mandatory	Data Type
name	Name	Yes	str
task_type	Type of a labeling job	No	LabelTaskTypeEnum

Parameter	Description	Mandatory	Data Type
delay	Whether the data object is configured when the phase is running. The default value is **False**.	No	bool

Example:

```
example = LabelTaskPlaceholder(name = "**")
# LabelTask object placeholder
```

Table 4-13 ServiceUpdatePlaceholder

Field	Description	Mandatory	Data Type
name	Name	Yes	str
delay	Whether the data object is configured when the phase is running. The default value is **False**.	No	bool

Example:

```
example = ServiceUpdatePlaceholder(name = "**")
# ServiceData object placeholder, which is used as the input for service update
```

Table 4-14 SWRImagePlaceholder

Field	Description	Mandatory	Data Type
name	Name	Yes	str
delay	Whether the data object is configured when the phase is running. The default value is **False**.	No	bool

Example:

```
example = SWRImagePlaceholder(name = "**" )
# SWRImage object placeholder, which is used as the input for model registration.
```

Table 4-15 ServiceInputPlaceholder

Parameter	Description	Mandatory	Data Type
name	Name	Yes	str

Parameter	Description	Mandatory	Data Type
model_name	Model name	Yes	str or Placeholder
model_versio n	Model version	No	str
envs	Environment variables	No	dict
delay	Whether service deployment information is configured when the phase is running. The default value is **True**.	No	bool

Example:

```
example = ServiceInputPlaceholder(name = "**" , model_name = "model_name")
# This object is used as the input for service deployment or service update.
```

Table 4-16 DataSelector

Parameter	Description	Mandatory	Data Type
name	Name	Yes	str
data_type_lis t	Supported data types. Currently, only **obs** and **dataset** are supported.	Yes	list
delay	Whether the data object is configured when the phase is running. The default value is **False**.	No	bool

Example:

```
example = DataSelector(name = "**" ,data_type_list=["obs", "dataset"])
# This object is used as the input of the job phase.
```

Table 4-17 DataConsumptionSelector

Parameter	Description	Mandatory	Data Type
data_list	Output data objects of a dependency phase	Yes	list

Example:

```
example = DataConsumptionSelector(data_list=[step1.outputs["step1_output_name"].as_input(),
step2.outputs["step2_output_name"].as_input()])
# Use the valid output from either step 1 or step 2 as the input. If step 1 is skipped and has no output, use
the valid output from step 2 as the input. (Make sure that data_list has only one valid output.)
```

4.1.4 Development State

In the development state, workflow Python SDKs are used to develop and debug workflows. This is a familiar and flexible development mode for AI developers and MLOps developers, which provides the following capabilities:

- Development and building: You can use Python code to create and orchestrate workflows with flexibility.

- Debugging: The debug and run modes are supported. The run mode supports partial execution and fully execution of a workflow.

- Publishing: The debugged workflows can be fixed and published to the running state for configuration and execution.

- Sharing: Workflows can be published to AI Gallery as assets and shared with other users.

4.1.5 Running State

Workflows run in a visualized mode, which is called the running state. You only need to pay attention to some simple parameter settings to start a workflow. Running workflows are released from the development state or subscribed to from AI Gallery.

A running workflow supports:

- Unified configuration management: The parameters and resources required for a workflow are centrally managed.

- Easy-to-use operations: You can start, stop, retry, copy, and delete workflows.

- Running record: records historical running parameters and statuses of the workflow.

4.2 Parameter Configuration

4.2.1 Function

A workflow parameter is a placeholder object that can be configured when the workflow runs. The following data types are supported: int, str, bool, float, Enum, dict, and list. You can display fields (such as algorithm hyperparameters) in a phase as placeholders in a transparent way. You can modify and use the default values that are set for them.

4.2.2 Parameter Overview

Placeholder

Parameter	Description	Mandatory	Data Type
name	Parameter name, which must be globally unique.	Yes	str
placeholder _type	Parameter type. The mapping between placeholder types and actual data types: PlaceholderType.INT -> int PlaceholderType.STR -> str PlaceholderType.BOOL -> bool PlaceholderType.FLOAT -> float PlaceholderType.ENUM -> Enum PlaceholderType.JSON -> dict PlaceholderType.LIST -> list ● When the type is PlaceholderType.ENUM, the **enum_list** field cannot be empty. ● When the type is PlaceholderType.LIST, the **placeholder_format** field cannot be empty and can only be set to **str**, **int**, **float**, or **bool**, indicating the data types in the list.	Yes	PlaceholderT ype
default	Default parameter value. The data type must be the same as that of **placeholder_type**.	No	Any
placeholder _format	Supported data formats. Currently, **obs**, **flavor**, **train_flavor**, and **swr** are supported.	No	str
delay	Whether parameters are set when the workflow is running. The default value is **False**, indicating that parameters are set before the workflow runs. If the value is **True**, parameters are set in an action of the phase where they are needed.	No	bool
description	Parameter description	No	str

Parameter	Description	Mandatory	Data Type
enum_list	List of enumerated values of a parameter. This parameter is mandatory only for parameters of PlaceholderType.ENUM type.	No	list
constraint	Constraints on parameters. This parameter only supports the constraints of training specifications and is not visible to you.	No	dict
required	Whether the parameter is mandatory. ● The default value is **True**. ● This parameter cannot be set to **False** for **Delay**. This parameter is optional at the frontend during execution.	No	bool

4.2.3 Examples

- Integer parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_int", placeholder_type=wf.PlaceholderType.INT, default=1,
  description="This is an integer parameter.")
  ```

- String parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_str", placeholder_type=wf.PlaceholderType.STR,
  default="default_value", description="This is a string parameter.")
  ```

- Bool parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_bool", placeholder_type=wf.PlaceholderType.BOOL, default=True,
  description="This is a bool parameter.")
  ```

- Float parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_float", placeholder_type=wf.PlaceholderType.FLOAT, default=0.1,
  description="This is a float parameter.")
  ```

- Enumeration parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_enum", placeholder_type=wf.PlaceholderType.ENUM, default="a",
  enum_list=["a", "b"], description="This is an enumeration parameter.")
  ```

- Dictionary parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_dict", placeholder_type=wf.PlaceholderType.JSON, default={"key":
  "value"}, description="This is a dictionary parameter.")
  ```

- List parameter
  ```
  from modelarts import workflow as wf
  wf.Placeholder(name="placeholder_list", placeholder_type=wf.PlaceholderType.LIST, default=[1, 2],
  placeholder_format="int", description="This is a list parameter and its value is an integer.")
  ```

4.3 Unified Storage

4.3.1 Function

Unified storage is used for workflow directory management. It centrally manages all storage paths of a workflow with these functions:

- Input directory management: When developing a workflow, you can centrally manage all data storage paths. You can store data and configure the root directory based on your own requirements. This function orchestrates directories but does not create them.

- Output directory management: When developing a workflow, you can centrally manage all output paths. You do not need to create output directories. Instead, you only need to configure the root path before the workflow runs and view the output data in the specified directories based on your directory orchestration rules. In addition, multiple executions of the same workflow are output to different directories, isolating data for different executions.

4.3.2 Common Usage

- **InputStorage** (path concatenation)

  This object is used to centrally manage input directories. The following is an example:

  ```
  import modelarts.workflow as wf
  storage = wf.data.InputStorage(name="storage_name", title="title_info",
  description="description_info") # Only name is mandatory.
  input_data = wf.data.OBSPath(obs_path = storage.join("directory_path")) # Add a slash (/) after a
  directory, for example, storage.join("/input/data/").
  ```

  When a workflow is running, if the root path of the storage object is **/root/**, the obtained path will be **/root/directory_path**.

- **OutputStorage** (directory creation)

  This object is used to centrally manage output directories and ensure that multiple executions of the same workflow are output to different directories. The following is an example:

  ```
  import modelarts.workflow as wf
  storage = wf.data.OutputStorage(name="storage_name", title="title_info",
  description="description_info") # Only name is mandatory.
  output_path = wf.data.OBSOutputConfig(obs_path = storage.join("directory_path")) # Only a
  directory can be created but not files.
  ```

  When a workflow is running, if the root path of the storage object is set to **/root/**, the system will automatically create a relative directory and the obtained path will be **/root/***Execution ID***/directory_path**.

4.3.3 Advanced Usage

Storage

This object contains capabilities of InputStorage and OutputStorage and can be flexibly used based on your needs.

Parameter	Description	Mandatory	Data Type
name	Name	Yes	str

Parameter	Description	Manda tory	Data Type
title	If this parameter is left blank, the value of **name** is used by default.	No	str
description	Description	No	str
create_dir	Whether to create a directory. The default value is **False**.	No	bool
with_executio n_id	Whether to combine **execution_id** when a directory is created. The default value is **False**. This parameter can be set to **True** only when **create_dir** is set to **True**.	No	bool

The following is an example:

- Implementing **InputStorage** capabilities
  ```
  import modelarts.workflow as wf
  # Create a Storage object (with_execution_id=False, create_dir=False).
  storage = wf.data.Storage(name="storage_name", title="title_info", description="description_info",
  with_execution_id=False, create_dir=False)
  input_data = wf.data.OBSPath(obs_path = storage.join("directory_path")) # Add a slash (/) after a
  directory, for example, storage.join("/input/data/").
  ```

  When a workflow is running, if the root path of the storage object is **/root/**, the obtained path will be **/root/directory_path**.

- Implementing **OutputStorage** capabilities
  ```
  import modelarts.workflow as wf
  # Create a Storage object (with_execution_id=True, create_dir=True).
  storage = wf.data.Storage(name="storage_name", title="title_info", description="description_info",
  with_execution_id=True, create_dir=True)
  output_path = wf.data.OBSOutputConfig(obs_path = storage.join("directory_path")) # Only a
  directory can be created.
  ```

  When a workflow is running, if the root path of the storage object is set to **/root/**, the system will automatically create a relative directory and the obtained path will be **/root/**_Execution ID_**/directory_path**.

- Implementing different capabilities of a Storage object through the **join** method
  ```
  import modelarts.workflow as wf
  # Create a Storage object. Assume that the root directory of the Storage object is /root/.
  storage = wf.data.Storage(name="storage_name", title="title_info", description="description_info",
  with_execution_id=False, create_dir=False)
  input_data1 = wf.data.OBSPath(obs_path = storage) # The obtained path is /root/.
  input_data2 = wf.data.OBSPath(obs_path = storage.join("directory_path")) # The obtained path is /
  root/directory_path. Ensure that the path exists.
  output_path1 = wf.data.OBSOutputConfig(obs_path = storage.join(directory="directory_path",
  with_execution_id=False, create_dir=True)) # The system automatically creates a directory /root/
  directory_path.
  output_path2 = wf.data.OBSOutputConfig(obs_path = storage.join(directory="directory_path",
  with_execution_id=True, create_dir=True)) # The system automatically creates a directory /root/
  Execution ID/directory_path.
  ```

Chain call is supported for **Storage**.

The following is an example:
```
import modelarts.workflow as wf
# Create a base class Storage object. Assume that the root directory of the Storage object is /root/.
```

```
storage = wf.data.Storage(name="storage_name", title="title_info", description="description_info",
with_execution_id=False, create_dir=Fals)
input_storage = storage.join("directory_path_1") # The obtained path is /root/directory_path_1.
input_storage_next = input_storage.join("directory_path_2") # The obtained path is /root/directory_path_1/
directory_path_2.
```

4.3.4 Example

Unified storage is mainly used in the job phase. The following code uses a workflow that contains only the training phase as an example.

```
from modelarts import workflow as wf

# Create an InputStorage object. Assume that the root directory of the Storage object is /root/input-data/.
input_storage = wf.data.InputStorage(name="input_storage_name", title="title_info",
description="description_info") # Only name is mandatory.

# Create an OutputStorage object. Assume that the root directory of the Storage object is /root/output/.
output_storage = wf.data.OutputStorage(name="output_storage_name", title="title_info",
description="description_info") # Only name is mandatory.

# Use JobStep to define a training phase, and set OBS paths for storing inputs and outputs.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name
    algorithm=wf.AIGalleryAlgorithm(subscription_id="subscription_ID",
item_version_id="item_version_ID"), # Algorithm used for training. In this example, an algorithm subscribed
to from AI Gallery is used.
    inputs=[
        wf.steps.JobInput(name="data_url_1", data=wf.data.OBSPath(obs_path = input_storage.join("/
dataset1/new.manifest"))),  # The obtained path is /root/input-data/dataset1/new.manifest.
        wf.steps.JobInput(name="data_url_2", data=wf.data.OBSPath(obs_path = input_storage.join("/
dataset2/new.manifest")))   # The obtained path is /root/input-data/dataset2/new.manifest.
    ],
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=output_storage.join("/model/"))), # The training output
path is /root/output/Execution ID/model/.
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
                flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")
        ),
        log_export_path=wf.steps.job_step.LogExportPath(obs_url=output_storage.join("/logs/"))  # The log
output path is /root/output/Execution ID/logs/.
    )# Training flavors
)

# Define a workflow that contains only the job phase.
workflow = wf.Workflow(
    name="test-workflow",
    desc="this is a test workflow",
    steps=[job_step],
    storages=[input_storage, output_storage] # Add Storage objects used in this workflow.
)
```

4.3.5 Operations

Configuring Root Paths in the Development State

Use the **run** method of the workflow object, and input root paths in the text box that is displayed when the workflow starts to run.

Figure 4-1 Inputting root paths

You must enter a valid path. If the path does not exist, an error will occur. The path format must be /*Bucket name*/*Folder path*/.

Configuring Root Paths in the Running State

Use the **release** method of the workflow object to release the workflow to the running state. On the ModelArts console, go to the **Workflow** page, find the target workflow, and configure root paths.

Figure 4-2 Configuring root paths

4.4 Phase Type

4.4.1 Dataset Creation Phase

4.4.1.1 Function

This phase integrates capabilities of the ModelArts dataset module, allowing you to create datasets of the new version. This phase is used to centrally manage existing data by creating datasets. It is usually followed by a dataset import phase or a labeling phase.

4.4.1.2 Parameter Overview

You can use CreateDatasetStep to create a dataset creation phase. The following is an example of defining a CreateDatasetStep.

Table 4-18 CreateDatasetStep

Parameter	Description	Mandatory	Data Type
name	Name of a dataset creation phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
inputs	Inputs of the dataset creation phase.	Yes	CreateDatasetInput or a list of CreateDatasetInput
outputs	Outputs of the dataset creation phase.	Yes	CreateDatasetOutput or a list of CreateDatasetOutput
properties	Configurations for dataset creation.	Yes	DatasetProperties
title	Title for frontend display.	No	str
description	Description of the dataset creation phase.	No	str
policy	Phase execution policy.	No	StepPolicy
depend_steps	Dependency phases.	No	Step or step list

Table 4-19 CreateDatasetInput

Parameter	Description	Mandatory	Data Type
name	Input name of the dataset creation phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str

Paramet er	Description	Man dato ry	Data Type
data	Input data object of the dataset creation phase.	Yes	OBS object. Currently, only OBSPath, OBSConsumption, OBSPlaceholder, and DataConsumption Selector are supported.

Table 4-20 CreateDatasetOutput

Paramet er	Description	Man dato ry	Data Type
name	Output name of the dataset creation phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str
config	Output configurations of the dataset creation phase.	Yes	Currently, only OBSOutputConfig is supported.

Table 4-21 DatasetProperties

Paramet er	Description	Man dat ory	Data Type
dataset_ name	Dataset name. The value contains 1 to 100 characters. Only letters, digits, underscores (_), and hyphens (-) are allowed.	Yes	str, Placeholder
dataset_f ormat	Dataset format. The default value is 0, indicating the file type.	No	**0**: file **1**: table
data_typ e	Data type. The default value is **FREE_FORMAT**.	No	DataTypeEnum

Paramet er	Description	Man dat ory	Data Type
descripti on	Description.	No	str
import_d ata	Whether to import data. The default value is **False**. Currently, only table data is supported.	No	bool
work_pat h_type	Type of the dataset output path. Currently, only OBS is supported. The default value is **0**.	No	int
import_c onfig	Configurations for label import. The default value is **None**. When creating a dataset based on labeled data, you can specify this parameter to import labeling information.	No	ImportConfig

Table 4-22 Importconfig

Parameter	Description	Man dato ry	Data Type
import_annotat ions	Whether to automatically import the labeling information in the input directory, supporting detection, image classification, and text classification. Options: ● **true**: The labeling information in the input directory is imported. (Default) ● **false**: The labeling information in the input directory is not imported.	No	str, Placeholder
import_type	Import mode. Options: ● **dir**: imported from an OBS path ● **manifest**: imported from a manifest file	No	**0**: file type ImportTypeEnum
annotation_for mat_config	Configurations of the imported labeling format	No	DAnnotationForm aTypeEtConumfig list

Table 4-23 AnnotationFormatConfig

Parameter	Description	Mandatory	Data Type
format_name	Name of a labeling format	No	AnnotationForma-tEnum
scene	Labeling scenario, which is optional	No	LabelTaskTypeEnum

Enumerated Type	Enumerated Value
ImportTypeEnum	DIR MANIFEST
DataTypeEnum	IMAGE TEXT AUDIO TABULAR VIDEO FREE_FORMAT
AnnotationFormatEnum	MA_IMAGE_CLASSIFICATION_V1 MA_IMAGENET_V1 MA_PASCAL_VOC_V1 YOLO MA_IMAGE_SEGMENTATION_V1 MA_TEXT_CLASSIFICATION_COMBINE_V1 MA_TEXT_CLASSIFICATION_V1 MA_AUDIO_CLASSIFICATION_DIR_V1

4.4.1.3 Examples

There are two scenarios:

- Creating a dataset using unlabeled data
- Creating a dataset using labeled data with labels imported

Creating a Dataset Using Unlabeled Data

Data preparation: Store unlabeled data in an OBS folder.

```
from modelarts import workflow as wf
# Use CreateDatasetStep to create a dataset of the new version using OBS data.

# Define parameters of the dataset output path.
```

```
dataset_output_path = wf.Placeholder(name="dataset_output_path",
placeholder_type=wf.PlaceholderType.STR, placeholder_format="obs")

# Define the dataset name.
dataset_name = wf.Placeholder(name="dataset_name", placeholder_type=wf.PlaceholderType.STR)

create_dataset = wf.steps.CreateDatasetStep(
    name="create_dataset", # Name of a dataset creation phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset creation", # Title, which defaults to the value of name
    inputs=wf.steps.CreateDatasetInput(name="input_name",
data=wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")),#
CreateDatasetStep inputs, configured when the workflow is running; the data field can also be represented
by the wf.data.OBSPath(obs_path="fake_obs_path") object.
    outputs=wf.steps.CreateDatasetOutput(name="output_name",
config=wf.data.OBSOutputConfig(obs_path=dataset_output_path)),# CreateDatasetStep outputs
    properties=wf.steps.DatasetProperties(
        dataset_name=dataset_name, # If the dataset name does not exist, a dataset will be created using
this name. If the dataset name exists, the corresponding dataset will be used.
        data_type=wf.data.DataTypeEnum.IMAGE, # Data type of the dataset, for example, image
    )
)
# Ensure that the dataset name is not used by others under the account. Otherwise, the dataset created by
others will be used in the subsequent phases.

workflow = wf.Workflow(
    name="create-dataset-demo",
    desc="this is a demo workflow",
    steps=[create_dataset]
)
```

Creating a Dataset Using Labeled Data with Labels Imported

Data preparation: Store labeled data in an OBS folder.

For details about specifications for importing labeled data from an OBS directory,
see **Specifications for Importing Data from an OBS Directory**.

```
from modelarts import workflow as wf
# Use CreateDatasetStep to create a dataset of the new version using OBS data.

# Define parameters of the dataset output path.
dataset_output_path = wf.Placeholder(name="dataset_placeholder_name",
placeholder_type=wf.PlaceholderType.STR, placeholder_format="obs")

# Define the dataset name.
dataset_name = wf.Placeholder(name="dataset_placeholder_name",
placeholder_type=wf.PlaceholderType.STR)

create_dataset = wf.steps.CreateDatasetStep(
    name="create_dataset", # Name of a dataset creation phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset creation", # Title, which defaults to the value of name
    inputs=wf.steps.CreateDatasetInput(name="input_name",
data=wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")),#
CreateDatasetStep inputs, configured when the workflow is running; the data field can also be represented
by the wf.data.OBSPath(obs_path="fake_obs_path") object.
    outputs=wf.steps.CreateDatasetOutput(name="output_name",
config=wf.data.OBSOutputConfig(obs_path=dataset_output_path)),# CreateDatasetStep outputs
    properties=wf.steps.DatasetProperties(
        dataset_name=dataset_name, # If the dataset name does not exist, a dataset will be created using
this name. If the dataset name exists, the corresponding dataset will be used.
        data_type=wf.data.DataTypeEnum.IMAGE, # Data type of the dataset, for example, image
        import_config=wf.steps.ImportConfig(
            annotation_format_config=[
                wf.steps.AnnotationFormatConfig(
```

```
                    format_name=wf.steps.AnnotationFormatEnum.MA_IMAGE_CLASSIFICATION_V1, # Labeling
format of labeled data
               scene=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION # Labeling scene
        ]
    )
  )
)
# Ensure that the dataset name is not used by others under the account. Otherwise, the dataset created by
others will be used in the subsequent phases.

workflow = wf.Workflow(
    name="create-dataset-demo",
    desc="this is a demo workflow",
    steps=[create_dataset]
)
```

4.4.2 Labeling Phase

4.4.2.1 Function

This phase integrates capabilities of the ModelArts dataset module, allowing you to label datasets. The labeling phase is used to create labeling jobs or label existing jobs.

4.4.2.2 Parameter Overview

You can use LabelingStep to create a labeling phase. The following is an example of defining a LabelingStep.

Table 4-24 LabelingStep

Parameter	Description	Mandatory	Data Type
name	Name of a labeling phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
inputs	Inputs of the labeling phase	Yes	LabelingInput or LabelingInput list
outputs	Outputs of the labeling phase	Yes	LabelingOutput or LabelingOutput list
properties	Configurations for dataset labeling	Yes	LabelTaskProper-ties

Parameter	Description	Mandatory	Data Type
title	Title for frontend display	No	str
description	Description of the labeling phase	No	str
policy	Phase execution policy	No	StepPolicy
depend_steps	Dependency phases	No	Step or step list

Table 4-25 LabelingInput

Parameter	Description	Mandatory	Data Type
name	Input name of the labeling phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str
data	Input data object of the labeling phase	Yes	Dataset or labeling job object. Currently, only Dataset, DatasetConsumption, DatasetPlaceholder, LabelTask, LabelTaskPlaceholder, LabelTaskConsumption, and DataConsumptionSelector are supported.

Table 4-26 LabelingOutput

Parameter	Description	Mandatory	Data Type
name	Output name of the labeling phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str

Table 4-27 LabelTaskProperties

Parameter	Description	Mandatory	Data Type
task_type	Type of a labeling job. Jobs of the specified type are returned.	Yes	LabelTaskTypeEnum
task_name	Labeling job name. The value contains 1 to 100 characters, including only letters, digits, hyphens (-), and underscores (_). This parameter is mandatory when the input is a dataset object.	No	str, Placeholder
labels	Labels to be created	No	Label
properties	Attributes of a labeling job. You can update this field to record custom information.	No	dict

Parameter	Description	Mandatory	Data Type
auto_sync_dataset	Whether to automatically synchronize the result of a labeling job to the dataset. Options: • **true**: The labeling result of the labeling job is automatically synchronized to the dataset. (Default) • **false**: The labeling result of the labeling job is not automatically synchronized to the dataset.	No	bool
content_labeling	Whether to enable content labeling for speech paragraph labeling. This function is enabled by default.	No	bool
description	Labeling job description. The description contains 0 to 256 characters and does not support the following special characters: ^!<>=&"'	No	str

Table 4-28 Label

Parameter	Description	Mandatory	Data Type
name	Tag name	No	str

Parameter	Description	Mandatory	Data Type
property	Basic attribute key-value pair of a label, such as color and shortcut keys	No	str, dic, Placeholder
type	Tag type	No	LabelTypeEnum

Enumerated Type	Enumerated Value
LabelTaskTypeEnum	IMAGE_CLASSIFICATION OBJECT_DETECTION IMAGE_SEGMENTATION TEXT_CLASSIFICATION NAMED_ENTITY_RECOGNITION TEXT_TRIPLE AUDIO_CLASSIFICATION SPEECH_CONTENT SPEECH_SEGMENTATION DATASET_TABULAR VIDEO_ANNOTATION FREE_FORMAT

4.4.2.3 Examples

There are three scenarios:

- Creating a labeling job for a specified dataset and labeling the dataset
- Labeling a specified job
- Creating a labeling job based on the output of the dataset creation phase

Creating a Labeling Job for a Specified Dataset and Labeling the Dataset

Scenarios:

- You have created only one unlabeled dataset and need to label it when the workflow is running.
- After a dataset is imported, the dataset needs to be labeled.

Data preparation: Create a dataset on the ModelArts console.

```
from modelarts import workflow as wf
# Use LabelingStep to create a labeling job for the input dataset and label it.

# Define the input dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")
```

```
# Define the name parameters of the labeling job.
task_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

labeling = wf.steps.LabelingStep(
    name="labeling", # Name of the labeling phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Dataset labeling", # Title, which defaults to the value of name
    properties=wf.steps.LabelTaskProperties(
        task_type=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION,   # Labeling job type, for example,
image classification
        task_name=task_name   # If the labeling job name does not exist, a job will be created using this
name. If the labeling job name exists, the corresponding job will be used.
    ),
    inputs=wf.steps.LabelingInput(name="input_name", data=dataset), # LabelingStep inputs. The dataset
object is configured when the workflow is running. You can also use
wf.data.Dataset(dataset_name="fake_dataset_name") for the data field.
    outputs=wf.steps.LabelingOutput(name="output_name"), # LabelingStep outputs
)

workflow = wf.Workflow(
    name="labeling-step-demo",
    desc="this is a demo workflow",
    steps=[labeling]
)
```

Labeling a Specified Job

Scenarios:

- You have created a labeling job and need to label it when the workflow is running.

- After a dataset is imported, the dataset needs to be labeled.

Data preparation: Create a labeling job using a specified dataset on the ModelArts console.

```
from modelarts import workflow as wf
# Input a labeling job and label it.

# Define the labeling job of the dataset.
label_task = wf.data.LabelTaskPlaceholder(name="label_task_placeholder_name")

labeling = wf.steps.LabelingStep(
    name="labeling", # Name of the labeling phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Dataset labeling", # Title, which defaults to the value of name
    inputs=wf.steps.LabelingInput(name="input_name", data=label_task), # LabelingStep inputs. The labeling
job object is configured when the workflow is running. You can also use
wf.data.LabelTask(dataset_name="dataset_name", task_name="label_task_name") for the data field.
    outputs=wf.steps.LabelingOutput(name="output_name"), # LabelingStep outputs
)

workflow = wf.Workflow(
    name="labeling-step-demo",
    desc="this is a demo workflow",
    steps=[labeling]
)
```

Creating a Labeling Phase Based on the Dataset Creation Phase

Scenario: The outputs of the dataset creation phase are used as the inputs of the labeling phase.

```
from modelarts import workflow as wf
```

```
# Define parameters of the dataset output path.
dataset_output_path = wf.Placeholder(name="dataset_output_path",
placeholder_type=wf.PlaceholderType.STR, placeholder_format="obs")

# Define the dataset name.
dataset_name = wf.Placeholder(name="dataset_name", placeholder_type=wf.PlaceholderType.STR)

create_dataset = wf.steps.CreateDatasetStep(
    name="create_dataset", # Name of a dataset creation phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset creation", # Title, which defaults to the value of name
    inputs=wf.steps.CreateDatasetInput(name="input_name",
data=wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")),#
CreateDatasetStep inputs, configured when the workflow is running; the data field can also be represented
by the wf.data.OBSPath(obs_path="fake_obs_path") object.
    outputs=wf.steps.CreateDatasetOutput(name="create_dataset_output",
config=wf.data.OBSOutputConfig(obs_path=dataset_output_path)),# CreateDatasetStep outputs
    properties=wf.steps.DatasetProperties(
        dataset_name=dataset_name, # If the dataset name does not exist, a dataset will be created using
this name. If the dataset name exists, the corresponding dataset will be used.
        data_type=wf.data.DataTypeEnum.IMAGE, # Data type of the dataset, for example, image
    )
)

# Define the name parameters of the labeling job.
task_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

labeling = wf.steps.LabelingStep(
    name="labeling", # Name of the labeling phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Dataset labeling", # Title, which defaults to the value of name
    properties=wf.steps.LabelTaskProperties(
        task_type=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION,   # Labeling job type, for example,
image classification
        task_name=task_name   # If the labeling job name does not exist, a job will be created using this
name. If the labeling job name exists, the corresponding job will be used.
    ),
    inputs=wf.steps.LabelingInput(name="input_name",
data=create_dataset.outputs["create_dataset_output"].as_input()), # LabelingStep inputs. The data source is
the outputs of the dataset creation phase.
    outputs=wf.steps.LabelingOutput(name="output_name"), # LabelingStep outputs
    depend_steps=create_dataset # Preceding dataset creation phase
)
# create_dataset is an instance of wf.steps.CreateDatasetStep. create_dataset_output is the name field
value of wf.steps.CreateDatasetOutput.

workflow = wf.Workflow(
    name="labeling-step-demo",
    desc="this is a demo workflow",
    steps=[create_dataset, labeling]
)
```

4.4.3 Dataset Import Phase

4.4.3.1 Function

This phase integrates capabilities of the ModelArts dataset module, allowing you
to import data to datasets. The dataset import phase is used to import data from
a specified path to a dataset or a labeling job. The application scenarios are as
follows:

- This phase is used for continuous data update. You can import raw data or
  labeled data to a labeling job and label the data in the labeling phase.

- Some labeled raw data can be directly imported to a dataset or labeling job, and the dataset with version information can be obtained in the dataset release phase.

4.4.3.2 Parameter Overview

You can use DatasetImportStep to create a dataset import phase. The following is an example of defining a DatasetImportStep.

Table 4-29 DatasetImportStep

Parameter	Description	Mandatory	Data Type
name	Name of a dataset import phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
inputs	Inputs of the dataset import phase.	Yes	DatasetImportInput or DatasetImportInput list
outputs	Outputs of the dataset import phase.	Yes	DatasetImportOutput or DatasetImportOutput list
properties	Configurations for dataset import.	Yes	ImportDataInfo
title	Title for frontend display.	No	str
description	Description of the dataset import phase.	No	str
policy	Phase execution policy.	No	StepPolicy
depend_steps	Dependency phases.	No	Step or step list

Table 4-30 DatasetImportInput

Parameter	Description	Mandatory	Data Type
name	Input name of the dataset import phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str
data	Input data object of the dataset import phase.	Yes	Dataset, OBS, or labeling job object. Currently, only Dataset, DatasetConsumption, DatasetPlaceholder, OBSPath, OBSConsumption, OBSPlaceholder, LabelTask, LabelTaskPlaceholder, LabelTaskConsumption, and DataConsumptionSelector are supported.

Table 4-31 DatasetImportOutput

Parameter	Description	Mandatory	Data Type
name	Output name of the dataset import phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str

Table 4-32 ImportDataInfo

Parameter	Description	Mandatory	Data Type
annotation_format_config	Configurations of the imported labeling format	No	AnnotationFormatConfig
excluded_labels	Samples with specified labels are not imported.	No	Label list

Parameter	Description	Mandatory	Data Type
import_annotated	Whether to import the labeled samples in the original dataset to the **To Be Confirmed** tab. The default value is **false**, indicating that the labeled samples in the original dataset are not imported to the **To Be Confirmed** tab. Options: ● **true**: The labeled samples in the original dataset are imported to the **To Be Confirmed** tab. ● **false**: The labeled samples in the original dataset are not imported to the **To Be Confirmed** tab.	No	bool
import_annotations	Whether to import labels. Options: ● **true**: The labels are imported. (Default) ● **false**: The labels are not imported.	No	bool
import_samples	Whether to import samples. Options: ● **true**: The samples are imported. (Default) ● **false**: The samples are not imported.	No	bool

Parameter	Description	Mandatory	Data Type
import_type	Import mode. Options: ● **dir**: imported from an OBS path ● **manifest**: imported from a manifest file	No	ImportTypeEnum
included_labels	Samples with specified labels are imported.	No	Label list
label_format	Label format. This parameter is used only for text datasets.	No	LabelFormat

Table 4-33 AnnotationFormatConfig

Parameter	Description	Mandatory	Data Type
format_name	Name of a labeling format	No	AnnotationForma-tEnum
parameters	Advanced parameters of the labeling format	No	AnnotationFormat Parameters
scene	Labeling scenario, which is optional	No	LabelTaskTypeEnum

Table 4-34 AnnotationFormatParameters

Parameter	Description	Mandatory	Data Type
difficult_only	Whether to import only hard examples. Options: ● **true**: Only hard examples are imported. ● **false**: All the samples are imported. (Default)	No	bool
included_labels	Samples with specified labels are imported.	No	Label list

Parameter	Description	Mandatory	Data Type
label_separator	Separator between labels. By default, the comma (,) is used as the separator. The separator needs to be escaped. The separator can contain only one character, which must be a letter, a digit, or any of the following special characters: !@#$%^&*_=|?/':.;,	No	str
sample_label_separator	Separator between the text and label. By default, the **Tab** key is used as the separator. The separator needs to be escaped. The separator can contain only one character, which must be a letter, a digit, or any of the following special characters: !@#$%^&*_=|?/':.;,	No	str

4.4.3.3 Examples

There are three scenarios:

- Importing data in a specified path to a target dataset
  - Importing labeled data to a dataset
  - Importing unlabeled data to a dataset
- Importing data in a specified path to a target labeling job
  - Importing labeled data to a labeling job
  - Importing unlabeled data to a labeling job
- Creating a dataset import phase based on the dataset creation phase

Importing Data in a Specified Path to a Target Dataset

Scenario: Data needs to be updated for a dataset.

- You import labeled data (with label information) in a specified path to a dataset. Then, you can create a dataset release phase to release a version.

  Data preparation: Create a dataset on the ModelArts console and upload labeled data to OBS.

  ```
  from modelarts import workflow as wf
  # Use DatasetImportStep to import data in a specified path to a dataset and output the dataset.

  # Define the dataset.
  dataset = wf.data.DatasetPlaceholder(name="input_dataset")
  ```

```
# Define the OBS data.
obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) #
object_type must be file or directory.

dataset_import = wf.steps.DatasetImportStep(
    name="data_import", # Name of the dataset import phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Dataset import", # Title, which defaults to the value of name
    inputs=[
        wf.steps.DatasetImportInput(name="input_name_1", data=dataset), # The target dataset is
configured when the workflow is running. You can also use
wf.data.Dataset(dataset_name="dataset_name") for the data field.
        wf.steps.DatasetImportInput(name="input_name_2", data=obs) # Storage path to the imported
dataset, configured when the workflow is running. You can also use
wf.data.OBSPath(obs_path="obs_path") for the data field.
    ],# DatasetImportStep inputs
    outputs=wf.steps.DatasetImportOutput(name="output_name"), # DatasetImportStep outputs
    properties=wf.steps.ImportDataInfo(
        annotation_format_config=[
            wf.steps.AnnotationFormatConfig(
                format_name=wf.steps.AnnotationFormatEnum.MA_IMAGE_CLASSIFICATION_V1, #
Labeling format of labeled data, for example, image classification
                scene=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION # Labeling scene
            )
        ]
    )
)

workflow = wf.Workflow(
    name="dataset-import-demo",
    desc="this is a demo workflow",
    steps=[dataset_import]
)
```

- You import unlabeled data in a specified path to a dataset. Then, you can add
  a labeling phase to label the imported data.

  Data preparation: Create a dataset on the ModelArts console and upload
  unlabeled data to OBS.

```
from modelarts import workflow as wf
# Use DatasetImportStep to import data in a specified path to a dataset and output the dataset.

# Define the dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Define the OBS data.
obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) #
object_type must be file or directory.

dataset_import = wf.steps.DatasetImportStep(
    name="data_import", # Name of the dataset import phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Dataset import", # Title, which defaults to the value of name
    inputs=[
        wf.steps.DatasetImportInput(name="input_name_1", data=dataset), # The target dataset is
configured when the workflow is running. You can also use
wf.data.Dataset(dataset_name="dataset_name") for the data field.
        wf.steps.DatasetImportInput(name="input_name_2", data=obs) # Storage path to the imported
dataset, configured when the workflow is running. You can also use
wf.data.OBSPath(obs_path="obs_path") for the data field.
    ],# DatasetImportStep inputs
    outputs=wf.steps.DatasetImportOutput(name="output_name"), # DatasetImportStep outputs
)

workflow = wf.Workflow(
    name="dataset-import-demo",
```

```
    desc="this is a demo workflow",
    steps=[dataset_import]
)
```

Importing Data in a Specified Path to a Target Labeling Job

Scenario: Data needs to be updated for a labeling job.

- You import labeled data in a specified path to a labeling job. Then, you can create a dataset release phase to release a version.

  Data preparation: Create a labeling job using a specified dataset and upload the labeled data to OBS.

  ```
  from modelarts import workflow as wf
  # Use DatasetImportStep to import data in a specified path to a labeling job and output the labeling job.

  # Define the labeling job.
  label_task = wf.data.LabelTaskPlaceholder(name="label_task_placeholder_name")

  # Define the OBS data.
  obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) # object_type must be file or directory.

  dataset_import = wf.steps.DatasetImportStep(
      name="data_import", # Name of the dataset import phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.
      title="Dataset import", # Title, which defaults to the value of name
      inputs=[
          wf.steps.DatasetImportInput(name="input_name_1", data=label_task), # Labeling job object, configured when the workflow is running. You can also use wf.data.LabelTask(dataset_name="dataset_name", task_name="label_task_name") for the data field.
          wf.steps.DatasetImportInput(name="input_name_2", data=obs) # Storage path to the imported dataset, configured when the workflow is running. You can also use wf.data.OBSPath(obs_path="obs_path") for the data field.
      ],# DatasetImportStep inputs
      outputs=wf.steps.DatasetImportOutput(name="output_name"), # DatasetImportStep outputs
      properties=wf.steps.ImportDataInfo(
          annotation_format_config=[
              wf.steps.AnnotationFormatConfig(
                  format_name=wf.steps.AnnotationFormatEnum.MA_IMAGE_CLASSIFICATION_V1, # Labeling format of labeled data, for example, image classification
                  scene=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION # Labeling scene
              )
          ]
      )
  )

  workflow = wf.Workflow(
      name="dataset-import-demo",
      desc="this is a demo workflow",
      steps=[dataset_import]
  )
  ```

- You import unlabeled data in a specified path to a labeling job. Then, you can add a labeling phase to label the imported data.

  Data preparation: Create a labeling job using a specified dataset and upload the unlabeled data to OBS.

  ```
  from modelarts import workflow as wf
  # Use DatasetImportStep to import data in a specified path to a labeling job and output the labeling job.

  # Define the labeling job.
  label_task = wf.data.LabelTaskPlaceholder(name="label_task_placeholder_name")
  ```

```
# Define the OBS data.
obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) #
object_type must be file or directory.

dataset_import = wf.steps.DatasetImportStep(
    name="data_import", # Name of the dataset import phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Dataset import", # Title, which defaults to the value of name
    inputs=[
        wf.steps.DatasetImportInput(name="input_name_1", data=label_task), # Labeling job object,
configured when the workflow is running. You can also use
wf.data.LabelTask(dataset_name="dataset_name", task_name="label_task_name") for the data
field.
        wf.steps.DatasetImportInput(name="input_name_2", data=obs) # Storage path to the imported
dataset, configured when the workflow is running. You can also use
wf.data.OBSPath(obs_path="obs_path") for the data field.
    ],# DatasetImportStep inputs
    outputs=wf.steps.DatasetImportOutput(name="output_name"), # DatasetImportStep outputs
)

workflow = wf.Workflow(
    name="dataset-import-demo",
    desc="this is a demo workflow",
    steps=[dataset_import]
)
```

Creating a Dataset Import Phase Based on the Dataset Creation Phase

Scenario: The outputs of the dataset creation phase are used as the inputs of the dataset import phase.

```
from modelarts import workflow as wf
# Use DatasetImportStep to import data in a specified path to a dataset and output the dataset.

# Define the OBS data.
obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) # object_type
must be file or directory.

dataset_import = wf.steps.DatasetImportStep(
    name="data_import", # Name of the dataset import phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset import", # Title, which defaults to the value of name
    inputs=[
        wf.steps.DatasetImportInput(name="input_name_1",
data=create_dataset.outputs["create_dataset_output"].as_input()), # The outputs of the dataset creation
phase are used as the inputs of the dataset import phase.
        wf.steps.DatasetImportInput(name="input_name_2", data=obs) # Storage path to the imported
dataset, configured when the workflow is running. You can also use
wf.data.OBSPath(obs_path="obs_path") for the data field.
    ],# DatasetImportStep inputs
    outputs=wf.steps.DatasetImportOutput(name="output_name"), # DatasetImportStep outputs
    depend_steps=create_dataset # Preceding dataset creation phase
)
# create_dataset is an instance of wf.steps.CreateDatasetStep. create_dataset_output is the name field
value of wf.steps.CreateDatasetOutput.

workflow = wf.Workflow(
    name="dataset-import-demo",
    desc="this is a demo workflow",
    steps=[dataset_import]
)
```

4.4.4 Dataset Release Phase

4.4.4.1 Function

This phase integrates capabilities of the ModelArts dataset module, enabling automatic dataset version release. The dataset release phase is used to release versions of existing datasets or labeling jobs. Each version is a data snapshot and can be used for subsequent data source tracing. The application scenarios are as follows:

- After data labeling is completed, a dataset version can be automatically released and used as inputs in subsequent phases.
- When data update is required for model training, you can use the dataset import phase to import data and then use the dataset release phase to release a version for subsequent phases.

4.4.4.2 Parameter Overview

You can use ReleaseDatasetStep to create a dataset release phase. The following is an example of defining a ReleaseDatasetStep.

Table 4-35 ReleaseDatasetStep

Parameter	Description	Mandatory	Data Type
name	Name of a dataset release phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
inputs	Inputs of the dataset release phase	Yes	ReleaseDatasetInput or ReleaseDatasetInput list
outputs	Outputs of the dataset release phase	Yes	ReleaseDatasetOutput or ReleaseDatasetOutput list
title	Title for frontend display	No	str
description	Description of the dataset release phase	No	str
policy	Phase execution policy	No	StepPolicy
depend_steps	Dependency phases	No	Step or step list

Table 4-36 ReleaseDatasetInput

Parameter	Description	Mandatory	Data Type
name	Input name of the dataset release phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str
data	Input data object of the dataset release phase	Yes	Dataset or labeling job object. Currently, only Dataset, DatasetConsumption, DatasetPlaceholder, LabelTask, LabelTaskPlacehold-er, LabelTaskConsump-tion, and DataConsumptionSe-lector are supported.

Table 4-37 ReleaseDatasetOutput

Parameter	Description	Mandatory	Data Type
name	Output name of the dataset release phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str
dataset_version_config	Configurations for dataset version release	Yes	DatasetVersionCon-fig

Table 4 **DatasetVersionConfig**

Parameter	Description	Mandatory	Data Type
version_na me	Dataset version name. By default, the dataset version is named in ascending order of V001 and V002.	No	str or Placeholder
version_for mat	Version format, which defaults to **Default**. You can also set it to **CarbonData**.	No	str
train_evalu ate_sample _ratio	Ratio between the training set and validation set, which defaults to **1.00**. The value ranges from 0 to 1.00. For example, **0.8** indicates the ratio for the training set is 80%, and that for the validation set is 20%.	No	str or Placeholder
clear_hard_ property	Whether to clear hard examples. The default value is **True**.	No	bool or Placeholder
remove_sa mple_usage	Whether to clear existing usage information of a dataset. The default value is **True**.	No	bool or Placeholder

Parameter	Description	Mandatory	Data Type
label_task_type	Type of a labeling job. If the input is a dataset, this field is mandatory and is used to specify the labeling scenario of the dataset version. If the input is a labeling job, this field does not need to be configured.	No	LabelTaskTypeEnum The following types are supported: ● IMAGE_CLASSIFICATION ● OBJECT_DETECTION = 1 ● IMAGE_SEGMENTATION ● TEXT_CLASSIFICATION ● NAMED_ENTITY_RECOGNITION ● TEXT_TRIPLE ● AUDIO_CLASSIFICATION ● SPEECH_CONTENT SPEECH_SEGMENTATION ● TABLE ● VIDEO_ANNOTATION
description	Description of a version	No	str

☐ **NOTE**

If there is no special requirement, use the default values.

4.4.4.3 Examples

There are three scenarios:

● Releasing a dataset version

● Releasing a labeling job version

● Releasing a version based on the output of the labeling phase

Releasing a Dataset Version

Scenario: When data in a dataset is updated, this phase can be used to release a dataset version for subsequent phases to use.

```
from modelarts import workflow as wf
# Use ReleaseDatasetStep to release a version of the input dataset and output the dataset with version
```

```
information.

# Define the dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Define the split ratio between the training set and validation set
train_ration = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR,
default="0.8")

release_version = wf.steps.ReleaseDatasetStep(
    name="release_dataset", # Name of the dataset release phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset version release", # Title, which defaults to the value of name
    inputs=wf.steps.ReleaseDatasetInput(name="input_name", data=dataset), # ReleaseDatasetStep inputs.
The dataset object is configured when the workflow is running. You can also use
wf.data.Dataset(dataset_name="dataset_name") for the data field.
    outputs=wf.steps.ReleaseDatasetOutput(
        name="output_name",
        dataset_version_config=wf.data.DatasetVersionConfig(
            label_task_type=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION,  # Labeling job type for
dataset version release
            train_evaluate_sample_ratio=train_ration # Split ratio between the training set and validation set
            )
    ) # ReleaseDatasetStep outputs
)

workflow = wf.Workflow(
    name="dataset-release-demo",
    desc="this is a demo workflow",
    steps=[release_version]
)
```

Releasing a Labeling Job Version

When data or labeling information of a labeling job is updated, this phase can be
used to release a dataset version for subsequent phases to use.

```
from modelarts import workflow as wf
# Use ReleaseDatasetStep to release a version of the input labeling job and output the dataset with version
information.

# Define the labeling job.
label_task = wf.data.LabelTaskPlaceholder(name="label_task_placeholder_name")

# Define the split ratio between the training set and validation set
train_ration = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR,
default="0.8")

release_version = wf.steps.ReleaseDatasetStep(
    name="release_dataset", # Name of the dataset release phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset version release", # Title, which defaults to the value of name
    inputs=wf.steps.ReleaseDatasetInput(name="input_name", data=label_task), # ReleaseDatasetStep inputs
The labeling job object is configured when the workflow is running. You can also use
wf.data.LabelTask(dataset_name="dataset_name", task_name="label_task_name") for the data field.
    outputs=wf.steps.ReleaseDatasetOutput(name="output_name",
dataset_version_config=wf.data.DatasetVersionConfig(train_evaluate_sample_ratio=train_ration)), # Split
ratio between the training set and validation set
)

workflow = wf.Workflow(
    name="dataset-release-demo",
    desc="this is a demo workflow",
    steps=[release_version]
)
```

Creating a Dataset Release Phase Based on the Labeling Phase

Scenario: The outputs of the labeling phase are used as the inputs of the dataset release phase.

```
from modelarts import workflow as wf
# Use ReleaseDatasetStep to release a version of the input labeling job and output the dataset with version information.

# Define the split ratio between the training set and validation set
train_ration = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR, default="0.8")

release_version = wf.steps.ReleaseDatasetStep(
    name="release_dataset", # Name of the dataset release phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.
    title="Dataset version release", # Title, which defaults to the value of name
    inputs=wf.steps.ReleaseDatasetInput(name="input_name", data=labeling_step.outputs["output_name"].as_input()), # ReleaseDatasetStep inputs
The labeling job object is configured when the workflow is running. You can also use
wf.data.LabelTask(dataset_name="dataset_name", task_name="label_task_name") for the data field.
    outputs=wf.steps.ReleaseDatasetOutput(name="output_name", dataset_version_config=wf.data.DatasetVersionConfig(train_evaluate_sample_ratio=train_ration)), # Split ratio between the training set and validation set
    depend_steps = [labeling_step] # Preceding labeling phase
)
# labeling_step is an instance object of wf.steps.LabelingStep and output_name is the value of the name field of wf.steps.LabelingOutput.

workflow = wf.Workflow(
    name="dataset-release-demo",
    desc="this is a demo workflow",
    steps=[release_version]
)
```

4.4.5 Job Phase

4.4.5.1 Function

This phase defines the algorithm, input, and output of a job to implement ModelArts job management for data processing, model training, and model evaluation. The application scenarios are as follows:

- Data preprocessing such as image enhancement and noise reduction
- Model training for object detection and image classification

4.4.5.2 Parameter Overview

You can use JobStep to create a job phase. The following is an example of defining a JobStep.

Table 4-38 JobStep

Parameter	Description	Mandatory	Data Type
name	Name of a job phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
algorithm	Algorithm object	Yes	BaseAlgorithm, Algorithm, AIGalleryAlgorithm
spec	Job specifications	Yes	JobSpec
inputs	Inputs of a job phase	Yes	JobInput or JobInput list
outputs	Outputs of a job phase	Yes	JobOutput or JobOutput list
title	Title for frontend display	No	str
description	Description of a job phase	No	str
policy	Phase execution policy	No	StepPolicy
depend_steps	Dependency phases	No	Step or step list

Table 4-39 JobInput

Parameter	Description	Mandatory	Data Type
name	Input name of the job phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str
data	Input data object of a job phase	Yes	Dataset or OBS object. Currently, only Dataset, DatasetPlaceholder, DatasetConsumption, OBSPath, OBSConsumption, OBSPlaceholder, and DataConsumption Selector are supported.

Table 4-40 JobOutput

Parameter	Description	Mandatory	Data Type
name	Output name of the job phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str

Parameter	Description	Mandatory	Data Type
obs_config	OBS output configuration	No	OBSOutputConfig
model_config	Model output configuration	No	ModelConfig
metrics_config	Metrics configuration	No	MetricsConfig

Table 4-41 OBSOutputConfig

Parameter	Description	Mandatory	Data Type
obs_path	Existing OBS directory	Yes	str, Placeholder, Storage
metric_file	Name of the file that stores metric information	No	str, Placeholder

Table 4-42 BaseAlgorithm

Parameter	Description	Mandatory	Data Type
id	Algorithm ID	No	str
subscription_id	Subscription ID of the subscribed algorithm	No	str
item_version_id	Version ID of the subscribed algorithm	No	str
code_dir	Code directory	No	str, Placeholder, Storage
boot_file	Boot file	No	str, Placeholder, Storage
command	Boot command	No	str, Placeholder
parameters	Algorithm hyperparameters	No	AlgorithmParameters list
engine	Information about the image used by the job	No	JobEngine
environments	Environment variables	No	dict

Table 4-43 Algorithm

Parameter	Description	Mandatory	Data Type
algorithm_id	Algorithm ID	Yes	str
parameters	Algorithm hyperparameters	No	AlgorithmParameters list

Table 4-44 AIGalleryAlgorithm

Parameter	Description	Mandatory	Data Type
subscription_id	Subscription ID of the subscribed algorithm	Yes	str
item_version_id	Version ID of the subscribed algorithm	Yes	str
parameters	Algorithm hyperparameters	No	AlgorithmParameters list

Table 4-45 AlgorithmParameters

Parameter	Description	Mandatory	Data Type
name	Name of an algorithm hyperparameter	Yes	str
value	Value of an algorithm hyperparameter	Yes	int, bool, float, str, Placeholder, Storage

Table 4-46 JobEngine

Parameter	Description	Mandatory	Data Type
engine_id	Image ID	No	str, Placeholder
engine_name	Image name	No	str, Placeholder
engine_version	Image version	No	str, Placeholder
image_url	Image URL	No	str, Placeholder

Table 4-47 JobSpec

Parameter	Description	Mandatory	Data Type
resource	Resource	Yes	JobResource
log_export_path	Log output path	No	LogExportPath
schedule_policy	Job scheduling policy	No	SchedulePolicy
volumes	Information about the file system mounted to the job	No	list[Volume]

Table 4-48 JobResource

Parameter	Description	Mandatory	Data Type
flavor	Resource specifications	Yes	Placeholder
node_count	Number of nodes. The default value is **1**. If there are multiple nodes, distributed training is supported.	No	int, Placeholder

Table 4-49 SchedulePolicy

Parameter	Description	Mandatory	Data Type
priority	Job scheduling priority. The value can only be **1**, **2**, or **3**, indicating low, medium, and high priorities, respectively.	Yes	int, Placeholder

Table 4-50 Volume

Parameter	Description	Mandatory	Data Type
nfs	NFS file system object	No	NFS

Table 4-51 NFS

Parameter	Description	Mandatory	Data Type
nfs_server_path	Service address of the NFS file system.	Yes	str, Placeholder
local_path	Path mounted to the container.	Yes	str, Placeholder
read_only	Indicates if the mount mode is set to read-only.	No	bool, Placeholder

4.4.5.3 Obtaining Resources

Before creating a job phase, perform the following operations to obtain supported training flavors and engines:

- Import packages.
  ```
  from modelarts.session import Session
  from modelarts.estimatorV2 import TrainingJob
  from modelarts.workflow.client.job_client import JobClient
  ```

- Initialize a session.
  ```
  # If you develop a workflow in a local IDEA, initialize a session as follows:
  # 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"]
  # Decrypt the information if it is encrypted.
  session = Session(
      access_key=__AK, # AK information of your account
      secret_key=__SK, # SK information of your account
      region_name="***", # Region to which your account belongs
      project_id="***" # Project ID of your account
  )

  # If you develop a workflow in a notebook environment, initialize a session as follows:
  session = Session()
  ```

- Obtain public resource pools.
  ```
  # Obtain the specification list of public resource pools.
  spec_list = TrainingJob(session).get_train_instance_types(session) # A list is returned. You can download it.
  print(spec_list)
  ```

- Obtain dedicated resource pools.
  ```
  # Obtain the list of running dedicated resource pools.
  pool_list = JobClient(session).get_pool_list() # A list of dedicated resource pools is returned.
  pool_id_list = JobClient(session).get_pool_id_list() # An ID list of dedicated resource pools is returned.
  The following lists the flavor IDs of dedicated resource pools. Select one as required.
      modelarts.pool.visual.xlarge (1 card)
      modelarts.pool.visual.2xlarge (2 cards)
      modelarts.pool.visual.4xlarge (4 cards)
      modelarts.pool.visual.8xlarge (8 cards)
  ```

- Obtain engine types.
  ```
  # Obtain engine types.
  engine_dict = TrainingJob(session).get_engine_list(session) # A dictionary is returned. You can
  ```

```
        download it.
        print(engine_dict)
```

4.4.5.4 Examples

There are seven scenarios:

- Using an algorithm subscribed to in AI Gallery
- Using an algorithm in Algorithm Management
- Using a custom algorithm (code directory+boot file+official image)
- Using a custom algorithm (code directory+boot command+official image)
- Creating a job phase based on the dataset release phase
- Job phase with visualization
- Using the DataSelector object as the input, which supports OBS or datasets

Using an Algorithm Subscribed to in AI Gallery

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") #
Only name is mandatory.

# Define the input dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id", # algorithm subscription ID. You can also enter the version number.
        item_version_id="item_version_id", # Algorithm version ID. You can also enter the version number
instead.
        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool,
float, or string.
        ]
    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If the
value of an algorithm hyperparameter does not need to be changed, you do not need to configure the
hyperparameter in parameters. Hyperparameter values will be automatically filled.

    inputs=wf.steps.JobInput(name="data_url", data=dataset), # JobStep inputs are configured when the
workflow is running. You can also use wf.data.Dataset(dataset_name="fake_dataset_name",
version_name="fake_version_name") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    )# Training flavors
)

workflow = wf.Workflow(
```

```
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[job_step],
    storages=[storage]
)
```

Using an algorithm in Algorithm Management

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") # Only name is mandatory.

# Define the input dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.Algorithm(
        algorithm_id="algorithm_id", # Algorithm ID
        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR, default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool, float, or string.
        ]
    ), # Algorithm used for training. An algorithm from Algorithm Management is used in this example. If the value of an algorithm hyperparameter does not need to be changed, you do not need to configure the hyperparameter in parameters. Hyperparameter values will be automatically filled.

    inputs=wf.steps.JobInput(name="data_url", data=dataset), # JobStep inputs are configured when the workflow is running. You can also use wf.data.Dataset(dataset_name="fake_dataset_name", version_name="fake_version_name") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON, description="Training flavor")
        )
    )# Training flavors
)

workflow = wf.Workflow(
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[job_step],
    storages=[storage]
)
```

Using a Custom Algorithm (Code Directory+Boot File+Official Image)

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") # Only name is mandatory.

# Define the input dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")
```

```
# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.BaseAlgorithm(
        code_dir="fake_code_dir", # Code directory
        boot_file="fake_boot_file", # Boot file path, which must be in the code directory
        engine=wf.steps.JobEngine(engine_name="fake_engine_name",
engine_version="fake_engine_version"), # Name and version of the official image

        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool,
float, or string.
        ]
    ), # The custom algorithm is implemented using the code directory, boot file, and official image.


    inputs=wf.steps.JobInput(name="data_url", data=dataset), # JobStep inputs are configured when the
workflow is running. You can also use wf.data.Dataset(dataset_name="fake_dataset_name",
version_name="fake_version_name") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    )# Training flavors
)

workflow = wf.Workflow(
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[job_step],
    storages=[storage]
)
```

Using a Custom Algorithm (Code Directory+Boot Command+Official Image)

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") #
Only name is mandatory.

# Define the input dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.BaseAlgorithm(
        code_dir="fake_code_dir", # Code directory
        command="fake_command", # Boot command
        engine=wf.steps.JobEngine(image_url="fake_image_url"), # Custom image URL, in the format of
Organization name/Image name:Version name. Do not contain the domain name; If image_url is required
to be configurable in the running state, use the following: image_url=wf.Placeholder(name="image_url",
placeholder_type=wf.PlaceholderType.STR, placeholder_format="swr", description="Custom image")
        parameters=[
```

```
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool,
float, or string.
        ]
    ), The custom algorithm is implemented using the code directory, boot command, and official image.

    inputs=wf.steps.JobInput(name="data_url", data=dataset), # JobStep inputs are configured when the
workflow is running. You can also use wf.data.Dataset(dataset_name="fake_dataset_name",
version_name="fake_version_name") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    )# Training flavors
)

workflow = wf.Workflow(
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[job_step],
    storages=[storage]
)
```

📖 **NOTE**

> The preceding four methods use a dataset as the input. If you want to use an OBS path as
> the input, set **data** of **JobInput** to
> **data=wf.data.OBSPlaceholder(name="obs_placeholder_name",
> object_type="directory")** or **data=wf.data.OBSPath(obs_path="fake_obs_path")**.
>
> In addition, you can specify a dataset or OBS path when creating a workflow to reduce
> configuration operations and facilitate debugging in the development state. You are advised
> to use placeholders to create a workflow you want to publish to the running state or AI
> Gallery. In this case, you can configure parameters before workflow execution.

Creating a Job Phase Based on the Dataset Release Phase

Scenario: The output of the dataset release phase is used as the input of the job
phase.

```
from modelarts import workflow as wf

# Define the dataset object.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Define the split ratio between the training set and validation set
train_ration = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR,
default="0.8")

release_version_step = wf.steps.ReleaseDatasetStep(
    name="release_dataset", # Name of the dataset release phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Dataset version release", # Title, which defaults to the value of name
    inputs=wf.steps.ReleaseDatasetInput(name="input_name", data=dataset), # ReleaseDatasetStep inputs.
The dataset object is configured when the workflow is running. You can also use
wf.data.Dataset(dataset_name="dataset_name") for the data field.
    outputs=wf.steps.ReleaseDatasetOutput(
        name="output_name",
        dataset_version_config=wf.data.DatasetVersionConfig(
            label_task_type=wf.data.LabelTaskTypeEnum.IMAGE_CLASSIFICATION,  # Labeling job type for
```

```
dataset version release
            train_evaluate_sample_ratio=train_ration # Split ratio between the training set and validation set
        )
    ) # ReleaseDatasetStep outputs
)

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") #
Only name is mandatory.

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id", # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id", # Version ID of the subscribed algorithm
        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool,
float, or string.
        ]
    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If the
value of an algorithm hyperparameter does not need to be changed, you do not need to configure the
hyperparameter in parameters. Hyperparameter values will be automatically filled.

    inputs=wf.steps.JobInput(name="data_url",
data=release_version_step.outputs["output_name"].as_input()), # The output of the dataset release phase is
used as the input of JobStep.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    ), # Training flavors
    depend_steps=release_version_step # Preceding dataset release phase
)
# release_version_step is an instance object of wf.steps.ReleaseDatasetStep and output_name is the
value of the name field of wf.steps.ReleaseDatasetOutput.

workflow = wf.Workflow(
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[release_version_step, job_step],
    storages=[storage]
)
```

Job Phase With Visualization

Phase visualization enables you to view the metrics generated by your workflows in real time. You can also display the external disks of each phase separately. To use phase visualization, you need to add and configure an output for showing metrics through the MetricsConfig object, based on the original job phase.

Table 4-52 MetricsConfig

Parameter	Description	Mandatory	Data Type
metric_files	Metric files	Yes	List. Elements in the list support string, placeholder, or storage.
realtime_visualization	Whether to display the output metrics in real time	No	Bool. The default value is **False**.
visualization	Whether to display visualization phases separately	No	Bool. The default value is **True**.

The output metrics file must contain standard JSON data with a maximum size of 1 MB. The data formats must match the supported ones.

- Key-value pair data

```
[
  {
    "key": "loss",
    "title": "loss",
    "type": "float",
    "data": {
      "value": 1.2
    }
  },
  {
    "key": "accuracy",
    "title": "accuracy",
    "type": "float",
    "data": {
      "value": 1.6
    }
  }
]
```

- Line chart data

```
[
  {
    "key": "metric",
    "title": "metric",
    "type": "line chart",
    "data": {
      "x_axis": [
        {
          "title": "step/epoch",
          "value": [
            1,
            2,
            3
          ]
        }
      ],
      "y_axis": [
```

```
            {
                "title": "value",
                "value": [
                    0.5,
                    0.4,
                    0.3
                ]
            }
        ]
    }
}
]
```

- Histogram data

```
[
    {
        "key": "metric",
        "title": "metric",
        "type": "histogram",
        "data": {
            "x_axis": [
                {
                    "title": "step/epoch",
                    "value": [
                        1,
                        2,
                        3
                    ]
                }
            ],
            "y_axis": [
                {
                    "title": "value",
                    "value": [
                        0.5,
                        0.4,
                        0.3
                    ]
                }
            ]
        }
    }
]
```

- Confusion matrix

```
[
    {
        "key": "confusion_matrix",
        "title": "confusion_matrix",
        "type": "table",
        "data": {
            "cell_value": [
                [
                    1,
                    2
                ],
                [
                    2,
                    3
                ]
            ],
            "col_labels": {
                "title": "labels",
                "value": [
                    "daisy",
                    "dandelion"
                ]
            },
            "row_labels": {
```

```
        "title": "predictions",
        "value": [
          "daisy",
          "dandelion"
        ]
      }
    }
  }
]
```

- One-dimensional table

```
[
  {
    "key": "Application Evaluation Results",
    "title": "Application Evaluation Results",
    "type": "one-dimensional-table",
    "data": {
      "cell_value": [
        [
          10,
          2,
          0.5
        ]
      ],
      "labels": [
        "samples",
        "maxResTine",
        "p99"
      ]
    }
  }
]
```

Example:

```
from modelarts import workflow as wf

# Create a Storage object to centrally manage training output directories.
storage = wf.data.Storage(name="storage_name", title="title_info", description="description_info",
with_execution_id=True, create_dir=True) # Only name is mandatory.

# Define the input dataset.
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id", # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id", # Algorithm version ID. You can also enter the version
number instead.
        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer,
bool, float, or string.
        ]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If
the value of an algorithm hyperparameter does not need to be changed, you do not need to
configure the hyperparameter in parameters. Hyperparameter values will be automatically filled.


    inputs=wf.steps.JobInput(name="data_url", data=dataset), # JobStep inputs are configured when
the workflow is running. You can also use wf.data.Dataset(dataset_name="fake_dataset_name",
```

```
version_name="fake_version_name") for the data field.
    outputs=[
    wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))),# JobStep outputs
        wf.steps.JobOutput(name="metrics_output",
metrics_config=wf.data.MetricsConfig(metric_files=storage.join("directory_path/metrics.json",
create_dir=False))) # Metrics are output to the configured path by the job script.
    ],
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    )# Training flavors
)

workflow = wf.Workflow(
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[job_step],
    storages=[storage]
)
```

◳ **NOTE**

Workflow does not automatically retrieve the metrics produced by training. You need to extract the metrics from the algorithm code, create the **metrics.json** file in the required data format, and upload the file to the OBS path specified in MetricsConfig. Workflow only reads, renders, and displays the data.

Using the DataSelector Object as the Input, Which Supports OBS or Datasets

You can use this method when you can choose the input type. The DataSelector object allows you to select either a dataset object or an OBS object as the training input. Here is a code example:

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") #
Only name is mandatory.

# Define the DataSelector object.
data_selector = wf.data.DataSelector(name="input_data", data_type_list=["dataset", "obs"])

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id", # algorithm subscription ID. You can also enter the version number.
        item_version_id="item_version_id", # Algorithm version ID. You can also enter the version number
instead.
        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool,
float, or string.
        ]
    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If the
value of an algorithm hyperparameter does not need to be changed, you do not need to configure the
hyperparameter in parameters. Hyperparameter values will be automatically filled.
```

```
    inputs=wf.steps.JobInput(name="data_url", data=data_selector), # JobStep inputs are configured when
the workflow is running. You can choose OBS or datasets as the input.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    )# Training flavors
)

workflow = wf.Workflow(
    name="job-step-demo",
    desc="this is a demo workflow",
    steps=[job_step],
    storages=[storage]
)
```

📖 **NOTE**

> When using DataSelector as the input, ensure that the algorithm input supports both datasets and OBS.

4.4.6 Model Registration Phase

4.4.6.1 Function

This phase integrates capabilities of ModelArts AI application management. This enables trained models to be registered in AI Application Management for service deployment and update. The application scenarios are as follows:

- Registering models trained from ModelArts training jobs
- Registering models from custom images

4.4.6.2 Parameter Overview

You can use ModelStep to create a model registration phase. The following is an example of defining a ModelStep.

Table 4-53 ModelStep

Parameter	Description	Mandatory	Data Type
name	Name of a model registration phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
inputs	Inputs of the model registration phase	No	ModelInput or ModelInput list

Parameter	Description	Mandatory	Data Type
outputs	Outputs of the model registration phase	Yes	ModelOutput or ModelOutput list
title	Title for frontend display	No	str
description	Description of the model registration phase	No	str
policy	Phase execution policy	No	StepPolicy
depend_steps	Dependency phases	No	Step or step list

Table 4-54 ModelInput

Parameter	Description	Mandatory	Data Type
name	Input name of the model registration phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str
data	Input data object of the model registration phase	Yes	OBS, SWR, or subscribed model object. Currently, only OBSPath, SWRImage, OBSConsumption, OBSPlaceholder, SWRImagePlaceholder, DataConsumptionSelector, and GalleryModel are supported.

Table 4-55 ModelOutput

Parame ter	Description	Manda tory	Data Type
name	Output name of the model registration phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str
model_c onfig	Configurations for model registration	Yes	ModelConfig

Table 4-56 ModelConfig

Paramete r	Description	Man dato ry	Data Type
model_typ e	Model type. Supported types: **TensorFlow**, **MXNet**, **Caffe**, **Spark_MLlib**, **Scikit_Learn**, **XGBoost**, **Image**, **PyTorch**, **Template**, and **Custom**. The default value is **TensorFlow**.	Yes	str
model_na me	Model name. Enter 1 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed.	No	str, Placeholder
model_ver sion	Model version in the format of *Digit.Digit.Digit*. The value range of the digits is [1, 99]. If this parameter is left blank, the version number automatically increases. **CAUTION** No part of the version number can start with 0. For example, **01.01.01** is not allowed.	No	str, Placeholder
runtime	Model runtime environment. The options of **runtime** are the same as those of **model_type**.	No	str, Placeholder
descriptio n	Model description that consists of 1 to 100 characters. The following special characters cannot be contained: &!'"<>=	No	str

Parameter	Description	Mandatory	Data Type
execution_code	OBS path for storing the execution code. By default, this parameter is left blank. The name of the execution code file is fixed to **customize_service.py**. The inference code file must be stored in the **model** directory. This parameter is left blank. The system can automatically identify the inference code in the **model** directory.	No	str
dependencies	Package required for the inference code and model. By default, this parameter is left blank. It is read from the configuration file.	No	str
model_metrics	Model precision, which is read from the configuration file	No	str
apis	All **apis** input and output parameters of a model (optional), which are parsed from the configuration file	No	str
initial_config	Model configuration information	No	dict
template	Template configuration items. This parameter is mandatory when **model_type** is set to **Template**.	No	Template
dynamic_load_mode	Dynamic loading mode. Currently, only **Single** is supported.	No	str, Placeholder
prebuild	Whether the model is prebuilt. The default value is **False**.	No	bool, Placeholder
install_type	Model installation type. The value can be **real_time**, **edge**, **batch**. If this parameter is left blank, all types are supported by default.	No	list[str]

Table 4-57 Template

Parameter	Description	Mandatory	Data Type
template_id	ID of the used template. The template has a built-in input and output mode.	Yes	str, Placeholder

Paramete r	Description	Mand atory	Data Type
infer_form at	Input and output mode ID. When this parameter is used, the input and output mode built in the template does not take effect.	No	str, Placeholder
template_i nputs	Template input configuration, specifying the source path for configuring a model	Yes	list of TemplateInputs object

Table 4-58 TemplateInputs

Paramete r	Description	Manda tory	Data Type
input_id	Input item ID, which is obtained from the template details	Yes	str, Placeholder
input	Template input path, which can be an OBS file path or OBS directory path. When you use a template with multiple input items to create a model, if the target paths **input_properties** specified in the template are the same, the OBS directory or OBS file name entered here must be unique to prevent files from being overwritten.	Yes	str, Placeholder, Storage

4.4.6.3 Examples

There are six scenarios:

- Registering models output by JobStep
- Registering a model using OBS data
- Registering a model using a template
- Registering a model using a custom image
- Registering a model using a custom image and OBS
- Registering a model using a subscribed model and OBS

Registering a Model From a Training Job (Model Source: JobStep Output)

```
import modelarts.workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info", description="description_info") #
Only name is mandatory.

# Define the input dataset object.
```

```
dataset = wf.data.DatasetPlaceholder(name="input_dataset")

# Use JobStep to define a training phase. Use a dataset as the input, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training", # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id", # algorithm subscription ID. You can also enter the version number.
        item_version_id="item_version_id", # Algorithm version ID. You can also enter the version number
instead.
        parameters=[
            wf.AlgorithmParameters(
                name="parameter_name",
                value=wf.Placeholder(name="parameter_name", placeholder_type=wf.PlaceholderType.STR,
default="fake_value",description="description_info")
            ) # Algorithm hyperparameters are represented using placeholders, which can be integer, bool,
float, or string.
        ]
    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If the
value of an algorithm hyperparameter does not need to be changed, you do not need to configure the
hyperparameter in parameters. Hyperparameter values will be automatically filled.

    inputs=wf.steps.JobInput(name="data_url", data=dataset), # JobStep inputs are configured when the
workflow is running. You can also use wf.data.Dataset(dataset_name="fake_dataset_name",
version_name="fake_version_name") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))), # JobStep outputs
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    )# Training flavors
)

# Define a model registration phase using ModelStep. The output of JobStep is used as the input of
ModelStep.

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_registration = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name='model_input', data=job_step.outputs["train_url"].as_input()),  # The
output of JobStep is used as the input of ModelStep.

outputs=wf.steps.ModelOutput(name='model_output',model_config=wf.steps.ModelConfig(model_name=mo
del_name, model_type="TensorFlow")), # ModelStep outputs
    depend_steps=job_step # Dependent job phase
)
# job_step is an instance object of wf.steps.JobStep and train_url is the value of the name field of
wf.steps.JobOutput.

workflow = wf.Workflow(
    name="model-step-demo",
    desc="this is a demo workflow",
    steps=[job_step, model_registration],
    storages=[storage]
)
```

Registering a Model From a Training Job (Model Source: A Trained Model Stored in OBS)

```
import modelarts.workflow as wf
# Define a model registration phase using ModelStep. The input is from OBS.

# Define the OBS data.
obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) # object_type
must be file or directory.

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_registration = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name='model_input', data=obs),  # ModelStep inputs are configured when
the workflow is running. You can also use wf.data.OBSPath(obs_path="fake_obs_path") for the data field.

outputs=wf.steps.ModelOutput(name='model_output',model_config=wf.steps.ModelConfig(model_name=mo
del_name, model_type="TensorFlow"))# ModelStep outputs
)

workflow = wf.Workflow(
    name="model-step-demo",
    desc="this is a demo workflow",
    steps=[model_registration]
)
```

Registering a Model Using a Template

```
import modelarts.workflow as wf
# Define a model registration phase using ModelStep. Register a model using a preset template.

# Define a preset template object. Fields in the template object can be represented by placeholders.
template = wf.steps.Template(
    template_id="fake_template_id",
    infer_format="fake_infer_format",
    template_inputs=[
        wf.steps.TemplateInputs(
            input_id="fake_input_id",
            input="fake_input_file"
        )
    ]
)

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_registration = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    outputs=wf.steps.ModelOutput(
        name='model_output',
        model_config=wf.steps.ModelConfig(
            model_name=model_name,
            model_type="Template",
            template=template
        )
    ) # ModelStep outputs
)

workflow = wf.Workflow(
    name="model-step-demo",
    desc="this is a demo workflow",
```

```
    steps=[model_registration]
)
```

Registering a Model From a Custom Image

```
import modelarts.workflow as wf
# Define a model registration phase using ModelStep. The input is from the URL of a custom image.

# Define the image data.
swr = wf.data.SWRImagePlaceholder(name="placeholder_name")

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_registration = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name="input",data=swr), # ModelStep inputs are configured when the
workflow is running. You can also use wf.data.SWRImage(swr_path="fake_path") for the data field.

outputs=wf.steps.ModelOutput(name='model_output',model_config=wf.steps.ModelConfig(model_name=mo
del_name, model_type="TensorFlow"))# ModelStep outputs
)

workflow = wf.Workflow(
    name="model-step-demo",
    desc="this is a demo workflow",
    steps=[model_registration]
)
```

Registering a Model Using a Custom Image and OBS

```
import modelarts.workflow as wf
# Define a model registration phase using ModelStep. The input is from the URL of a custom image.

# Define the image data.
swr = wf.data.SWRImagePlaceholder(name="placeholder_name")

# Define OBS model data.
model_obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) #
object_type must be file or directory.

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_registration = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    inputs=[
        wf.steps.ModelInput(name="input",data=swr), # ModelStep inputs are configured when the workflow
is running. You can also use wf.data.SWRImage(swr_path="fake_path") for the data field.
        wf.steps.ModelInput(name="input",data=model_obs) # ModelStep inputs are configured when the
workflow is running. You can also use wf.data.OBSPath(obs_path="fake_obs_path") for the data field.
    ],
    outputs=wf.steps.ModelOutput(
        name='model_output',
        model_config=wf.steps.ModelConfig(
            model_name=model_name,
            model_type="Custom",
            dynamic_load_mode="Single"
        )
    ) # ModelStep outputs
)

workflow = wf.Workflow(
```

```
        name="model-step-demo",
        desc="this is a demo orkflow",
        steps=[model_registration]
)
```

Registering a Model Using a Subscribed Model and OBS

This mode is similar to the custom image + OBS mode, except that you obtain a custom image from a subscribed model.

Example:

```
import modelarts.workflow as wf

# Define the subscribed model object.
base_model = wf.data.GalleryModel(subscription_id="fake_subscription_id", version_num="fake_version") #
Model subscribed to from AI Gallery, generally published by a developer

# Define OBS model data.
model_obs = wf.data.OBSPlaceholder(name = "obs_placeholder_name", object_type = "directory" ) #
object_type must be file or directory.

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_registration = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    inputs=[
        wf.steps.ModelInput(name="input",data=base_model) # Use a subscribed model as the ModelStep
input.
        wf.steps.ModelInput(name="input",data=model_obs) # ModelStep inputs are configured when the
workflow is running. You can also use wf.data.OBSPath(obs_path="fake_obs_path") for the data field.
    ],
    outputs=wf.steps.ModelOutput(
        name='model_output',
        model_config=wf.steps.ModelConfig(
            model_name=model_name,
            model_type="Custom",
            dynamic_load_mode="Single"
        )
    ) # ModelStep outputs
)

workflow = wf.Workflow(
    name="model-step-demo",
    desc="this is a demo workflow",
    steps=[model_registration]
)
```

In the preceding example, the system automatically obtains the custom image from the subscribed model and registers and generates a model based on the entered OBS model path. **model_obs** can be replaced with the dynamic output of JobStep.

◯ NOTE

The value of **model_type** can be **TensorFlow**, **MXNet**, **Caffe**, **Spark_MLlib**, **Scikit_Learn**, **XGBoost**, **Image**, **PyTorch**, **Template**, or **Custom**.

If **model_type** is not set for **wf.steps.ModelConfig**, **TensorFlow** is used by default.

- If the model type of your workflow does not need to be changed, refer to the preceding examples.

- If the model type of your workflow needs to be changed in multiple executions, write the parameter using placeholders.

  model_type = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.ENUM, default="TensorFlow", enum_list=["TensorFlow", "MXNet", "Caffe", "Spark_MLlib", "Scikit_Learn", "XGBoost", "Image", "PyTorch", "Template", "Custom"], description="Model type")

4.4.7 Service Deployment Phase

4.4.7.1 Function

This phase integrates capabilities of ModelArts service management to enable service deployment and update in a workflow. The application scenarios are as follows:

- Deploying a model as a web service
- Updating an existing service (gray update supported)

4.4.7.2 Parameter Overview

You can use ServiceStep to create a service deployment phase. The following is an example of defining a ServiceStep.

Table 4-59 ServiceStep

Parameter	Description	Mandatory	Data Type
name	Name of a service deployment phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
inputs	Inputs of the service deployment phase	No	ServiceInput or ServiceInput list
outputs	Outputs of the service deployment phase	Yes	ServiceOutput or ServiceOutput list
title	Title for frontend display	No	str
description	Description of the service deployment phase	No	str
policy	Phase execution policy	No	StepPolicy
depend_steps	Dependency phases	No	Step or step list

Table 4-60 ServiceInput

Parameter	Description	Mandatory	Data Type
name	Input name of the service deployment phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The input name of a step must be unique.	Yes	str
data	Input data object of the service deployment phase	Yes	Model list or service object. Currently, only ServiceInputPlace-holder, ServiceData, and ServiceUpdatePla-ceholder are supported.

Table 4-61 ServiceOutput

Parameter	Description	Mandatory	Data Type
name	Output name of the service deployment phase. The name can contain a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-), and must start with a letter. The output name of a step must be unique.	Yes	str
service_config	Configurations for service deployment	Yes	ServiceConfig

Table 4 ServiceConfig

Parameter	Description	Mandatory	Data Type
infer_type	Inference mode. The value can be **real-time**, **batch**, or **edge**. The default value is **real-time**. • **real-time**: real-time service. The model is deployed as a web service. • **batch**: batch service. A batch service can perform inference on batch data and automatically stops after data processing is completed. • **edge**: edge service. A model is deployed as a web service on an edge node through IEF. Create an edge node on IEF beforehand.	Yes	str
service_name	Service name. Enter 1 to 64 characters. Only letters, digits, hyphens (-), and underscores (_) are allowed. **NOTE** If you do not specify this parameter, the default service name is generated automatically.	No	str, Placeholder
description	Service description, which contains a maximum of 100 characters. By default, this parameter is left blank.	No	str
vpc_id	ID of the VPC to which a real-time service instance is deployed. By default, this parameter is left blank. In this case, ModelArts allocates a dedicated VPC to each user, and users are isolated from each other. To access other service components in the VPC of the service instance, set this parameter to the ID of the corresponding VPC. Once a VPC is configured, it cannot be modified. If both **vpc_id** and **cluster_id** are configured, only the dedicated resource pool takes effect.	No	str

Parameter	Description	Mandatory	Data Type
subnet_network_id	ID of a subnet. By default, this parameter is left blank. This parameter is mandatory when **vpc_id** is configured. Enter the network ID displayed in the subnet details on the VPC management console. A subnet provides dedicated network resources that are isolated from other networks.	No	str
security_group_id	Security group. By default, this parameter is left blank. This parameter is mandatory when **vpc_id** is configured. A security group is a virtual firewall that provides secure network access control policies for service instances. A security group must contain at least one inbound rule to permit the requests whose protocol is TCP, source address is **0.0.0.0/0**, and port number is **8080**.	No	str
cluster_id	ID of a dedicated resource pool. By default, this parameter is left blank, indicating that no dedicated resource pool is used. When using a dedicated resource pool to deploy services, ensure that the cluster is running properly. After this parameter is configured, the network configuration of the cluster is used, and the **vpc_id** parameter does not take effect. If both this parameter and **cluster_id** in **real-time config** are configured, **cluster_id** in **real-time config** is preferentially used.	No	str
additional_properties	Additional configurations	No	dict
apps	Whether to enable application authentication for service deployment. Multiple application names can be entered.	No	str, Placeholder, list
envs	Environment variables	No	dict

Example:

```
example = ServiceConfig()
# This object is used in the output of the service deployment phase.
```

If there is no special requirement, use the default values.

4.4.7.3 Examples

There are three scenarios:

- Deploying a real-time service
- Modifying a real-time service
- Getting the inference address from the service deployment phase

Deploying a Real-Time Service

```
import modelarts.workflow as wf
# Use ServiceStep to define a service deployment phase and specify a model for service deployment.

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

service_step = wf.steps.ServiceStep(
    name="service_step", # Name of the service deployment phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Deploying a service",  # Title
    inputs=wf.steps.ServiceInput(name="si_service_ph",
data=wf.data.ServiceInputPlaceholder(name="si_placeholder1",
                                        # Restrictions on the model name: Only the model
name specified here can be used in the running state; use the same model name as model_name of the
model registration phase.
                                        model_name=model_name)),# ServiceStep inputs
    outputs=wf.steps.ServiceOutput(name="service_output") # ServiceStep outputs
)

workflow = wf.Workflow(
    name="service-step-demo",
    desc="this is a demo workflow",
    steps=[service_step]
)
```

Modifying a Real-Time Service

Scenario: When you use a new model version to update an existing service, ensure that the name of the new model version is the same as that of the deployed service.

```
import modelarts.workflow as wf
# Use ServiceStep to define a service deployment phase and specify a model for service update.

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

# Define a service object.
service = wf.data.ServiceUpdatePlaceholder(name="placeholder_name")

service_step  = wf.steps.ServiceStep(
    name="service_step", # Name of the service deployment phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Service update", # Title
    inputs=[wf.steps.ServiceInput(name="si2",
data=wf.data.ServiceInputPlaceholder(name="si_placeholder2",
                                        # Restrictions on the model name: Only the model
```

```
name specified here can be used in the running state.
                                                     model_name=model_name)),
        wf.steps.ServiceInput(name="si_service_data", data=service) # ServiceStep inputs are configured
when the workflow is running. You can also use wf.data.ServiceData(service_id="fake_service") for the
data field.
    ], # ServiceStep inputs
    outputs=wf.steps.ServiceOutput(name="service_output") # ServiceStep outputs
)

workflow = wf.Workflow(
    name="service-step-demo",
    desc="this is a demo workflow",
    steps=[service_step]
)
```

Getting the Inference Address From the Service Deployment Phase

The service deployment phase supports the output of the inference address. You can use the **get_output_variable("access_address")** method to obtain the output and use it in subsequent phases.

- For services deployed in the public resource pool, you can use **access_address** to obtain the inference address registered on the public network from the output.

- For services deployed in a dedicated resource pool, you can get the internal inference address from the output using **cluster_inner_access_address**, in addition to the public inference address. The internal address can only be accessed by other inference services.

```
import modelarts.workflow as wf

# Define model name parameters.
sub_model_name = wf.Placeholder(name="si_placeholder1",
placeholder_type=wf.PlaceholderType.STR)

sub_service_step = wf.steps.ServiceStep(
    name="sub_service_step", # Name of the service deployment phase. The name contains a
maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must
start with a letter and must be unique in a workflow.
    title="Subservice",  # Title
    inputs=wf.steps.ServiceInput(
        name="si_service_ph",
        data=wf.data.ServiceInputPlaceholder(name="si_placeholder1", model_name=sub_model_name)
    ),# ServiceStep inputs
    outputs=wf.steps.ServiceOutput(name="service_output") # ServiceStep outputs
)


main_model_name = wf.Placeholder(name="si_placeholder2",
placeholder_type=wf.PlaceholderType.STR)

# Obtain the inference address output by the subservice and transfer the address to the main service
through envs.
main_service_config = wf.steps.ServiceConfig(
    infer_type="real-time",
    envs={"infer_address":
sub_service_step.outputs["service_output"].get_output_variable("access_address")} # Obtain the
inference address output by the subservice and transfer the address to the main service through envs.
)

main_service_step = wf.steps.ServiceStep(
    name="main_service_step", # Name of the service deployment phase. The name contains a
maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must
start with a letter and must be unique in a workflow.
    title="Main service",  # Title
    inputs=wf.steps.ServiceInput(
        name="si_service_ph",
```

```
        data=wf.data.ServiceInputPlaceholder(name="si_placeholder2",
model_name=main_model_name)
    ),# ServiceStep inputs
    outputs=wf.steps.ServiceOutput(name="service_output", service_config=main_service_config), #
ServiceStep outputs
    depend_steps=sub_service_step
)

workflow = wf.Workflow(
    name="service-step-demo",
    desc="this is a demo workflow",
    steps=[sub_service_step, main_service_step]
)
```

4.4.7.4 Configuration Operations

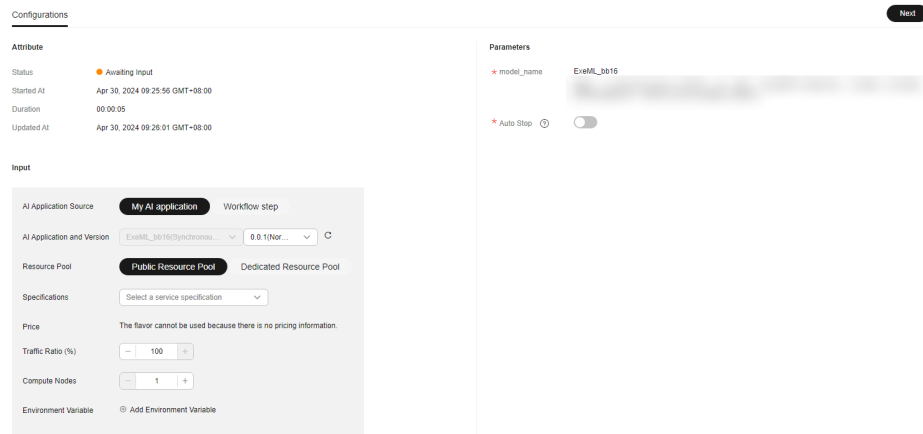### Configuring Information for Deploying a Synchronous Service

After the service deployment phase is started in the development state (usually a notebook instance), configure the information based on the following format in the logs.

1. On the ModelArts management console, choose **Workflow** from the left navigation pane.

2. Configure the information after the service deployment phase is started. After the configuration, click **Next**.

Configuring Information for Deploying an Asynchronous Service

1. On the ModelArts management console, choose **Workflow** from the left navigation pane.

2. Configure the information after the service deployment phase is started. Select an asynchronous inference AI application and a version, and configure service startup parameters. After the configuration. click **Next**.

   ☐ NOTE

   After you select the required AI application and version, the system automatically matches the service startup parameters.

4.4.8 Condition Phase

4.4.8.1 Function

This phase is used for conditional branching in the execution of phases based on condition value comparison or metrics output by the preceding phase. The application scenarios are as follows:

- You need to determine the subsequent process based on different input values. If you need to determine whether to retrain or register a model based on the model precision output by the training phase, you can use this phase to control the process.

4.4.8.2 Parameter Overview

You can use ConditionStep to create a condition phase. The following is an example of defining a ConditionStep.

Table 4-62 ConditionStep

Parameter	Description	Mandatory	Data Type
name	Name of a condition phase. The name contains a maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be unique in a workflow.	Yes	str
conditions	List of conditions. The AND operation is used for multiple conditions.	Yes	Condition or condition list
if_then_steps	Steps to be executed if the calculation result of the condition expression is **True**	No	str or str list
else_then_steps	Steps to be executed if the calculation result of the condition expression is **False**	No	str or str list
title	Title for frontend-phase display	No	str
description	Description of a condition phase	No	str
depend_steps	Dependency phases	No	Step or step list

Table 4-63 Condition

Parameter	Description	Mandatory	Data Type
condition_type	Condition type. The "==", ">", ">=", "in", "<", "<=", "!=", and "or" operators are supported.	Yes	ConditionTypeEnum
left	Left value of a condition expression	Yes	int, float, str, bool, Placeholder, Sequence, Condition, MetricInfo
right	Right value of a condition expression	Yes	int, float, str, bool, Placeholder, Sequence, Condition, MetricInfo

Table 4-64 MetricInfo

Parameter	Description	Mandatory	Data Type
input_data	Metric input. Currently, only the output of JobStep is supported.	Yes	JobStep output
json_key	Key value corresponding to the metric information to be obtained	Yes	str

Description of the structure:

- Condition object, which consists of the condition type, left value, and right value
  - The condition type is obtained from ConditionTypeEnum. The "==", ">", ">=", "in", "<", "<=", "!=", and "or" operators are supported. The following table describes the mapping.

Enumerated Type	Operator
ConditionTypeEnum.EQ	==
ConditionTypeEnum.GT	>
ConditionTypeEnum.GTE	>=
ConditionTypeEnum.IN	in

Enumerated Type	Operator
ConditionTypeEnum.LT	<
ConditionTypeEnum.LTE	<=
ConditionTypeEnum.NOT	!=
ConditionTypeEnum.OR	or

- – The left and right values support the following types: integer, float, string, bool, placeholder, sequence, condition, and MetricInfo.
- – A condition phase supports a list of condition objects. The && operation is performed between multiple conditions.
- if_then_steps and else_then_steps
  - – if_then_steps indicates a list of phases that are ready for execution if conditions evaluate to **true**. In this case, steps in else_then_steps are skipped.
  - – else_then_steps indicates a list of phases that are ready for execution if conditions evaluate to **false**. In this case, steps in if_then_steps are skipped.

4.4.8.3 Examples

Simple Examples

- Implemented using parameter configurations

```
import modelarts.workflow as wf

left_value = wf.Placeholder(name="left_value", placeholder_type=wf.PlaceholderType.BOOL,
default=True)

# Condition object
condition = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.EQ, left=left_value,
right=True) # Condition object, including the type, left value, and right value.

# Condition phase
condition_step = wf.steps.ConditionStep(
    name="condition_step_test", # Name of the condition phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    conditions=condition, # Condition objects. The relationship between the conditions is &&.
    if_then_steps="job_step_1", # If conditions evaluate to true, job_step_1 is ready for execution, and
job_step_2 is skipped.
    else_then_steps="job_step_2" # If conditions evaluate to false, job_step_2 is ready for execution,
and job_step_1 is skipped.
)

# This phase is used only as an example. You need to supplement other fields as required.
job_step_1 = wf.steps.JobStep(
    name="job_step_1",
    depend_steps=condition_step
)

# This phase is used only as an example. You need to supplement other fields as required.
model_step_1 = wf.steps.ModelStep(
    name="model_step_1",
    depend_steps=job_step_1
)
```

```
# This phase is used only as an example. You need to supplement other fields as required.
job_step_2 = wf.steps.JobStep(
    name="job_step_2",
    depend_steps=condition_step
)

# This phase is used only as an example. You need to supplement other fields as required.
model_step_2 = wf.steps.ModelStep(
    name="model_step_2",
    depend_steps=job_step_2
)

workflow = wf.Workflow(
    name="condition-demo",
    desc="this is a demo workflow",
    steps=[condition_step, job_step_1, job_step_2, model_step_1, model_step_2]
)
```

◫ **NOTE**

Scenario description: **job_step_1** and **job_step_2** indicate two training phases that depend on **condition_step**. condition_step parameters determine the subsequent phase execution.

Execution analysis:

– If the default value of **left_value** is **True**, the calculation result of the condition logical expression is **True**. Then, **job_step_1** is executed, **job_step_2** is skipped, and all phases contained in the branches that use **job_step_2** as the unique root node are skipped. That is, **model_step_2** is skipped. Therefore, **condition_step**, **job_step_1**, and **model_step_1** are executed.

– If **left_value** is set to **False**, the calculation result of the condition logical expression is **False**. Then, **job_step_2** is executed, **job_step_1** is skipped, and all phases contained in the branches that use **job_step_1** as the unique root node are skipped. That is, **model_step_1** is skipped, and **condition_step**, **job_step_2**, and **model_step_2** are executed.

● Implemented by obtaining the metric information output by JobStep

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.Storage(name="storage_name", title="title_info", with_execution_id=True,
create_dir=True, description="description_info")  # The name field is mandatory, and the title and
description fields are optional.

# Define the input OBS object.
obs_data = wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")

# Use JobStep to define a training phase, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Image classification training",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version
number instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If
the value of an algorithm hyperparameter does not need to be changed, you do not need to
configure the hyperparameter in parameters. Hyperparameter values will be automatically filled.
    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    outputs=[
```

```
wf.steps.JobOutput(name="train_url",obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("dir
ectory_path"))),
        wf.steps.JobOutput(name="metrics",
metrics_config=wf.data.MetricsConfig(metric_files=storage.join("directory_path/metrics.json",
create_dir=False))) # Metric output path. Metric information is automatically output by the job script
based on the specified data format. (In the example, the metric information needs to be output to the
metrics.json file in the training output directory.)
    ],
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")
        )
    ) # Training flavors
)


# Define a condition object.
condition_lt = wf.steps.Condition(
    condition_type=wf.steps.ConditionTypeEnum.LT,
    left=wf.steps.MetricInfo(job_step.outputs["metrics"].as_input(), "accuracy"),
    right=0.5
)


condition_step = wf.steps.ConditionStep(
    name="condition_step_test", # Name of the condition phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    conditions=condition_lt, # Condition objects. The relationship between the conditions is &&.
    if_then_steps="training_job_retrain", # If conditions evaluate to true, training_job_retrain is ready
for execution, and model_registration is skipped.
    else_then_steps="model_registration", # If conditions evaluate to false, model_registration is
ready for execution, and training_job_retrain is skipped.
    depend_steps=job_step
)


# Use JobStep to define a training phase, and use OBS to store the output.
job_step_retrain = wf.steps.JobStep(
    name="training_job_retrain",  # Name of a training phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Image classification retraining",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version
number instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If
the value of an algorithm hyperparameter does not need to be changed, you do not need to
configure the hyperparameter in parameters. Hyperparameter values will be automatically filled.
    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    outputs=[

wf.steps.JobOutput(name="train_url",obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("dir
ectory_path_retrain"))),
        wf.steps.JobOutput(name="metrics",
metrics_config=wf.data.MetricsConfig(metric_files=storage.join("directory_path_retrain/metrics.json",
create_dir=False))) # Metric output path. Metric information is automatically output by the job script
based on the specified data format. (In the example, the metric information needs to be output to the
metrics.json file in the training output directory.)
    ],
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor_retrain",
placeholder_type=wf.PlaceholderType.JSON, description="Training flavor")
        )
    ), # Training flavors
    depend_steps=condition_step
```

```
)

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_step = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a
maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must
start with a letter and must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name='model_input', data=job_step.outputs["train_url"].as_input()),  #
job_step output is used as the input.
    outputs=wf.steps.ModelOutput(name='model_output',
model_config=wf.steps.ModelConfig(model_name=model_name, model_type="TensorFlow")),  #
ModelStep outputs
    depend_steps=condition_step,
)

workflow = wf.Workflow(
    name="condition-demo",
    desc="this is a demo workflow",
    steps=[job_step, condition_step, job_step_retrain, model_step],
    storages=storage
)
```

In this example, ConditionStep obtains the accuracy output by **job_step** and compares it with the preset value to determine whether to retrain or register the model. When the accuracy output by **job_step** is less than the threshold 0.5, the calculation result of **condition_lt** is **True**. In this case, **job_step_retrain** runs and **model_step** skips. Otherwise, **job_step_retrain** skips and **model_step** runs.

📖 **NOTE**

> For details about the format requirements of the metric file generated by **job_step**, see **Job Phase**. In the condition phase, only the metric data whose type is float can be used as the input.

The following is an example of the **metrics.json** file:

```
[
    {
        "key": "loss",
        "title": "loss",
        "type": "float",
        "data": {
            "value": 1.2
        }
    },
    {
        "key": "accuracy",
        "title": "accuracy",
        "type": "float",
        "data": {
            "value": 0.8
        }
    }
]
```

Advanced Example

```
import modelarts.workflow as wf

left_value = wf.Placeholder(name="left_value", placeholder_type=wf.PlaceholderType.BOOL, default=True)
condition1 = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.EQ, left=left_value, right=True)

internal_condition_1 = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.GT, left=10, right=9)
internal_condition_2 = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.LT, left=10, right=9)
```

```
# The result of condition2 is internal_condition_1 || internal_condition_2.
condition2 = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.OR, left=internal_condition_1,
right=internal_condition_2)

condition_step = wf.steps.ConditionStep(
    name="condition_step_test", # Name of the condition phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    conditions=[condition1, condition2], # Condition objects. The relationship between the conditions is &&.
    if_then_steps=["job_step_1"], # If conditions evaluate to true, job_step_1 is ready for execution, and
job_step_2 is skipped.
    else_then_steps=["job_step_2"] # If conditions evaluate to false, job_step_2 is ready for execution, and
job_step_1 is skipped.
)

# This phase is used only as an example. You need to supplement other fields as required.
job_step_1 = wf.steps.JobStep(
    name="job_step_1",
    depend_steps=condition_step
)

# This phase is used only as an example. You need to supplement other fields as required.
job_step_2 = wf.steps.JobStep(
    name="job_step_2",
    depend_steps=condition_step
)

workflow = wf.Workflow(
    name="condition-demo",
    desc="this is a demo workflow",
    steps=[condition_step, job_step_1, job_step_2],
)
```

ConditionStep supports nested condition phases. You can flexibly design tit based on different scenarios.

NOTE

The condition phase can only support two branches, which is very limiting. You can use the new branch function to replace the ConditionStep capability without creating new phases. For details, see **Branch Control**.

4.5 Branch Control

Function

You can use parameters or metrics from training output to decide whether to run a phase. This way, you can control the process.

Application Scenarios

This function is used for complex scenarios that involve multiple branches. When each execution starts, the workflow decides which branches to run and which ones to skip based on the relevant configuration information. This way, only some branches are executed. This function has a similar use case as ConditionStep, but it is more powerful. This function applies to the dataset creation phase, labeling phase, dataset import phase, dataset release phase, job phase, model registration phase, and service deployment phase.

Examples

- **Controlling the execution of a single phase**
  - Implemented using parameter configurations
    ```
    from modelarts import workflow as wf

    condition_equal = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.EQ,
    left=wf.Placeholder(name="is_skip", placeholder_type=wf.PlaceholderType.BOOL), right=True)

    # Create an OutputStorage object to centrally manage training output directories.
    storage = wf.data.OutputStorage(name="storage_name", title="title_info",
                            description="description_info")  # Only name is mandatory.

    # Define the input OBS object.
    obs_data = wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")

    # Use JobStep to define a training phase, and use OBS to store the output.
    job_step = wf.steps.JobStep(
        name="training_job", # Name of a training phase. The name contains a maximum of 64
    characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a
    letter and must be unique in a workflow.
        title="Image classification training",  # Title, which defaults to the value of name.
        algorithm=wf.AIGalleryAlgorithm(
            subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
            item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version
    number instead.
            parameters=[]

        ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this
    example. If the value of an algorithm hyperparameter does not need to be changed, you do not
    need to configure the hyperparameter in parameters. Hyperparameter values will be
    automatically filled.

        inputs=wf.steps.JobInput(name="data_url", data=obs_data),
        # JobStep input is configured when the workflow is running. You can also use
    data=wf.data.OBSPath(obs_path="fake_obs_path") for the data field.
        outputs=wf.steps.JobOutput(name="train_url",

    obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))),
        # JobStep output
        spec=wf.steps.JobSpec(
            resource=wf.steps.JobResource(
                flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
    description="Training flavor")

            )
        ), # Training flavors
        policy=wf.steps.StepPolicy(
            skip_conditions=[condition_equal] # Determines whether to skip job_step based on the
    calculation result of skip_conditions.
        )
    )

    workflow = wf.Workflow(
        name="new-condition-demo",
        desc="this is a demo workflow",
        steps=[job_step],
        storages=storage
    )
    ```

    In this example, **job_step** has a skip policy that is controlled by a bool
    parameter. If the placeholder parameter named **is_skip** is set to **True**,
    then **job_step** is skipped when **condition_equal** evaluates to **True**.
    Otherwise, **job_step** is run. For more details about the condition object,
    see **Condition Phase**.
  - Implemented by obtaining the metric information output by JobStep

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.Storage(name="storage_name", title="title_info", with_execution_id=True,
create_dir=True, description="description_info")  # The name field is mandatory, and the title
and description fields are optional.

# Define the input OBS object.
obs_data = wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")

# Use JobStep to define a training phase, and use OBS to store the output.
job_step = wf.steps.JobStep(
    name="training_job", # Name of a training phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a
letter and must be unique in a workflow.
    title="Image classification training",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version
number instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this
example. If the value of an algorithm hyperparameter does not need to be changed, you do not
need to configure the hyperparameter in parameters. Hyperparameter values will be
automatically filled.
    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    outputs=[

wf.steps.JobOutput(name="train_url",obs_config=wf.data.OBSOutputConfig(obs_path=storage.joi
n("directory_path"))),
        wf.steps.JobOutput(name="metrics",
metrics_config=wf.data.MetricsConfig(metric_files=storage.join("directory_path/metrics.json",
create_dir=False))) # Metric output path. Metric information is automatically output by the job
script based on the specified data format. (In the example, the metric information needs to be
output to the metrics.json file in the training output directory.)
    ],
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")
        )
    )  # Training flavors
)

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name",
placeholder_type=wf.PlaceholderType.STR)

# Define a condition object.
condition_lt = wf.steps.Condition(
    condition_type=wf.steps.ConditionTypeEnum.LT,
    left=wf.steps.MetricInfo(job_step.outputs["metrics"].as_input(), "accuracy"),
    right=0.5
)

model_step = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a
maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It
must start with a letter and must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name='model_input',
data=job_step.outputs["train_url"].as_input()),  # job_step output is used as the input.
    outputs=wf.steps.ModelOutput(name='model_output',
model_config=wf.steps.ModelConfig(model_name=model_name, model_type="TensorFlow")),  #
ModelStep outputs
    depend_steps=job_step # Preceding job phase
    policy=wf.steps.StepPolicy(skip_conditions=condition_lt) # Determines whether to skip
model_step based on the calculation result of skip_conditions.
```

```
)

workflow = wf.Workflow(
    name="new-condition-demo",
    desc="this is a demo workflow",
    steps=[job_step, model_step],
    storages=storage
)
```

In this example, **model_step** has a skip policy. The model registration depends on whether the accuracy output by **job_step** meets the preset value. When the accuracy output by **job_step** is less than the threshold 0.5, the calculation result of **condition_lt** is **True**. In this case, **model_step** skips. Otherwise, **model_step** runs.

📖 NOTE

For details about the format requirements of the metric file generated by **job_step**, see **Branch Control**. In the condition phase, only the metric data whose type is float can be used as the input.

The following is an example of the **metrics.json** file:

```
[
    {
        "key": "loss",
        "title": "loss",
        "type": "float",
        "data": {
            "value": 1.2
        }
    },
    {
        "key": "accuracy",
        "title": "accuracy",
        "type": "float",
        "data": {
            "value": 0.8
        }
    }
]
```

- **Controlling partial execution of multiple branches**

```
from modelarts import workflow as wf

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.Storage(name="storage_name", title="title_info", with_execution_id=True,
create_dir=True, description="description_info")  # The name field is mandatory, and the title and
description fields are optional.

# Define the input OBS object.
obs_data = wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")

condition_equal_a = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.EQ,
left=wf.Placeholder(name="job_step_a_is_skip", placeholder_type=wf.PlaceholderType.BOOL),
right=True)

# Use JobStep to define a training phase, and use OBS to store the output.
job_step_a = wf.steps.JobStep(
    name="training_job_a",  # Name of a training phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Image classification training",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version
number instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If
```

the value of an algorithm hyperparameter does not need to be changed, you do not need to configure the hyperparameter in **parameters**. Hyperparameter values will be automatically filled.
```
    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    outputs=[wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path_a")))],
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    ),  # Training flavors
    policy=wf.steps.StepPolicy(skip_conditions=condition_equal_a)
)

condition_equal_b = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.EQ,
left=wf.Placeholder(name="job_step_b_is_skip", placeholder_type=wf.PlaceholderType.BOOL),
right=True)

# Use JobStep to define a training phase, and use OBS to store the output.
job_step_b = wf.steps.JobStep(
    name="training_job_b",  # Name of a training phase. The name contains a maximum of 64
characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter
and must be unique in a workflow.
    title="Image classification training",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version
number instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If
the value of an algorithm hyperparameter does not need to be changed, you do not need to
configure the hyperparameter in parameters. Hyperparameter values will be automatically filled.
    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    outputs=[wf.steps.JobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path_b")))],
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    ),  # Training flavors
    policy=wf.steps.StepPolicy(skip_conditions=condition_equal_b)
)

# Define model name parameters.
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_step = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a
maximum of 64 characters, including only letters, digits, underscores (_), and hyphens (-). It must
start with a letter and must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name='model_input',
data=wf.data.DataConsumptionSelector(data_list=[job_step_a.outputs["train_url"].as_input(),
job_step_b.outputs["train_url"].as_input()])),  # Select the output of job_step_a or job_step_b as the
input.
    outputs=wf.steps.ModelOutput(name='model_output',
model_config=wf.steps.ModelConfig(model_name=model_name, model_type="TensorFlow")),  #
ModelStep outputs
    depend_steps=[job_step_a, job_step_b],  # Preceding job phase
)

workflow = wf.Workflow(
    name="new-condition-demo",
    desc="this is a demo workflow",
    steps=[job_step_a, job_step_b, model_step],
```

```
    storages=storage
)
```

In this example, both **job_step_a** and **job_step_b** have a skip policy that is controlled by parameters. When the parameter values are different, the execution of **model_step** can be divided into the following cases (**model_step** has no skip policy configured, so it follows the default rule).

job_step_a_is_skip	job_step_b_is_skip	Whether to Execute model_step
True	True	No
	False	Yes
False	True	Yes
	False	Yes

⚠ CAUTION

Default rule: A phase is automatically skipped if all the phases it depends on are skipped. Otherwise, the phase is run. This logic can apply to any phase.

Based on the previous example, if you want to override the default rule and make **model_step** run when **job_step_a** and **job_step_b** are skipped, you only need to configure a skip policy in **model_step**. The skip policy takes precedence over the default rule.

4.6 Data Selection Among Multiple Inputs

Function

This function is only for the scenario where multiple branches are run. When you create a workflow phase, the data input source of the phase is uncertain. The data input source could be the output of any of the phases it depends on. Only after all dependency phases are run, the valid output is automatically selected as the input based on the actual execution situation.

Examples

```
from modelarts import workflow as wf

condition_equal = wf.steps.Condition(condition_type=wf.steps.ConditionTypeEnum.EQ,
left=wf.Placeholder(name="is_true", placeholder_type=wf.PlaceholderType.BOOL), right=True)
condition_step = wf.steps.ConditionStep(
    name="condition_step",
    conditions=[condition_equal],
    if_then_steps=["training_job_1"],
    else_then_steps=["training_job_2"],
)

# Create an OutputStorage object to centrally manage training output directories.
storage = wf.data.OutputStorage(name="storage_name", title="title_info",
                    description="description_info")  # Only name is mandatory.
```

```
# Define the input OBS object.
obs_data = wf.data.OBSPlaceholder(name="obs_placeholder_name", object_type="directory")

# Use JobStep to define a training phase, and use OBS to store the output.
job_step_1 = wf.steps.JobStep(
    name="training_job_1",  # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version number
instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If the
value of an algorithm hyperparameter does not need to be changed, you do not need to configure the
hyperparameter in parameters. Hyperparameter values will be automatically filled.

    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    # JobStep input is configured when the workflow is running. You can also use
data=wf.data.OBSPath(obs_path="fake_obs_path") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
                        obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))),
    # JobStep output
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    ), # Training flavors
    depend_steps=[condition_step]
)

# Use JobStep to define a training phase, and use OBS to store the output.
job_step_2 = wf.steps.JobStep(
    name="training_job_2",  # Name of a training phase. The name contains a maximum of 64 characters,
including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and must be
unique in a workflow.
    title="Image classification training",  # Title, which defaults to the value of name.
    algorithm=wf.AIGalleryAlgorithm(
        subscription_id="subscription_id",  # Subscription ID of the subscribed algorithm
        item_version_id="item_version_id",  # Algorithm version ID. You can also enter the version number
instead.
        parameters=[]

    ), # Algorithm used for training. An algorithm subscribed to in AI Gallery is used in this example. If the
value of an algorithm hyperparameter does not need to be changed, you do not need to configure the
hyperparameter in parameters. Hyperparameter values will be automatically filled.

    inputs=wf.steps.JobInput(name="data_url", data=obs_data),
    # JobStep input is configured when the workflow is running. You can also use
data=wf.data.OBSPath(obs_path="fake_obs_path") for the data field.
    outputs=wf.steps.JobOutput(name="train_url",
                        obs_config=wf.data.OBSOutputConfig(obs_path=storage.join("directory_path"))),
    # JobStep output
    spec=wf.steps.JobSpec(
        resource=wf.steps.JobResource(
            flavor=wf.Placeholder(name="train_flavor", placeholder_type=wf.PlaceholderType.JSON,
description="Training flavor")

        )
    ), # Training flavors
    depend_steps=[condition_step]
)

# Define model name parameters.
```

```
model_name = wf.Placeholder(name="placeholder_name", placeholder_type=wf.PlaceholderType.STR)

model_step = wf.steps.ModelStep(
    name="model_registration", # Name of the model registration phase. The name contains a maximum of
64 characters, including only letters, digits, underscores (_), and hyphens (-). It must start with a letter and
must be unique in a workflow.
    title="Model registration",  # Title
    inputs=wf.steps.ModelInput(name='model_input',
data=wf.data.DataConsumptionSelector(data_list=[job_step_1.outputs["train_url"].as_input(),
job_step_2.outputs["train_url"].as_input()])),  # Select the output of job_step_1 or job_step_2 as the input.
    outputs=wf.steps.ModelOutput(name='model_output',
model_config=wf.steps.ModelConfig(model_name=model_name, model_type="TensorFlow")), # ModelStep
outputs
    depend_steps=[job_step_1, job_step_2] # Preceding job phase
)# job_step is an instance object of wf.steps.JobStep and train_url is the value of the name field of
wf.steps.JobOutput.


workflow = wf.Workflow(name="data-select-demo",
                   desc="this is a test workflow",
                   steps=[condition_step, job_step_1, job_step_2, model_step],
                   storages=storage
                   )
```

☐ NOTE

> The workflow in this example has two parallel branches, but only one branch runs at a time, depending on the configuration of **condition_step**. The input source of **model_step** is either **job_step_1** or **job_step_2**'s output. If **job_step_1** runs and **job_step_2** is skipped, **model_step** uses **job_step_1**'s output as input, and vice versa.

4.7 Creating a Workflow

To create a workflow, define each phase by referring to **Phase Type**. You need to perform the following steps:

1. Sort out scenarios, understand preset steps' functions, and determine the DAG structure.
2. Debug single-phase functions like training or inference on ModelArts.
3. Choose the code template that matches the phase function and fill in the details.
4. Arrange phases according to the DAG structure to create a workflow.

4.8 Debugging a Workflow

After creating a workflow, you can debug it in the development state. You can use the run or debug modes for debugging. Suppose a workflow has five phases: **label_step**, **release_step**, **job_step**, **model_step**, and **service_step**. The debugging steps are as follows:

*   Run mode

    –   Running all phases
        ```
        workflow.run(steps=[label_step, release_step, job_step, model_step, service_step],
        experiment_id="Experiment record ID")
        ```

    –   Running **job_step**, **model_step**, and **service_step** (Ensure the correctness of data dependency for partial execution.)
        ```
        workflow.run(steps=[job_step, model_step, service_step], experiment_id="Experiment record ID")
        ```

- Debug mode

  You can use the debug mode only in a notebook environment with the **next()** method. Example:

  a. Start the debug mode.
     ```
     workflow.debug(steps=[label_step, release_step, job_step, model_step, service_step],
     experiment_id="Experiment record ID")
     ```

  b. Run the first phase.
     ```
     workflow.next()
     ```

     As a result, either of the following situations occur:

     - If the data required for running the phase is available, the phase will be executed directly.

     - If the data required for running the phase is unavailable, configure phase data in either of the following ways:

       ○ Configuring a parameter:
          ```
          workflow.set_placeholder ("Parameter name", Value)
          ```

       ○ Configuring a data object:
          ```
          workflow.set_data (Name of the data object, Data object)
          # Example: Configuring a dataset object.
          workflow.set_data("Object name", Dataset(dataset_name="Dataset name",
          version_name="Dataset version name"))
          ```

  c. After the execution on the preceding phase is completed, use **workflow.next()** to start the next phase. Then, repeat the operations until all phases have been executed.

     ◫ NOTE

     When you debug a workflow in the development state, the system only monitors the running status and prints logs. To view the detailed running information of each phase, go to the ModelArts console.

4.9 Publishing a Workflow

4.9.1 Publishing a Workflow to the Running State

After debugging a workflow, you can use the **release()** method to publish the workflow to the running state for configuration and execution (on the workflow page of the management console).

Run the following command:

```
workflow.release()
```

After the preceding command is executed, if the log indicates that the workflow is published, you can go to the ModelArts workflow page to view the workflow. For details about workflow operations, see **How to Use a Workflow?**.

The **release_and_run()** method is based on the **release()** method and allows you to publish and run workflows in the development state, without the need to configure and execute workflows on the console.

⚠ CAUTION

Note the following when using this method:

- For all configuration objects related to placeholders in the workflow, you need to either set default values or use fixed data objects directly.

- The method executes differently depending on the workflow object's name. It creates and runs a new workflow if the name does not exist. It updates and runs the existing workflow if the name already exists, using the new workflow structure for the new execution.

workflow.release_and_run()

4.9.2 Publishing a Workflow to AI Gallery

You can publish workflows to AI Gallery and share them with other users. To do so, run this code:

workflow.release_to_gallery()

Once the release is done, you can visit AI Gallery to see the asset details. The asset permission is set to private by default, but you can change it if you want.

The **release_to_gallery()** method contains the following input parameters.

Parameter	Description	Mandatory	Type
content_id	Workflow asset ID	No	str
version	Version number of the workflow asset. The format is *x.x.x*.	No	str
desc	Description of the workflow asset version	No	str
title	Workflow asset name. If this parameter is not specified, the workflow name is used by default.	No	str
visibility	Visibility of the workflow asset. The value can be **public**, **group** (whitelist), and **private** (visible only to you). The default value is **private**.	No	str
group_users	Whitelist. This parameter is mandatory only when visibility is set to **group**. You can only enter **domain_id**.	No	list[str]

You can use the method in two ways based on the input parameters:

- workflow.release_to_gallery(title="*Asset name*") publishes a new workflow asset with version 1.0.0. If the workflow includes an algorithm that is not from AI Gallery, it also publishes the algorithm to AI Gallery with version 1.0.0.

- workflow.release_to_gallery(content_id=" ***", title="*Asset name*") publishes a new version based on the specified workflow asset. The version number increases automatically. If the workflow includes an AI Gallery algorithm, it updates the algorithm asset with a higher version number.

Workflow asset whitelist setting:

You can specify the **visibility** and **group_users** fields of the **release_to_gallery** method when you publish an asset for the first time. To change the whitelist for accessing a certain asset, use this command:

```
from modelarts import workflow as wf

# Add specified whitelist users.
wf.add_whitelist_users(content_id="**", version_num="*.*.*", user_groups=["**", "**"])

# Delete specified whitelist users.
wf.delete_whitelist_users(content_id="**", version_num="*.*.*", user_groups=["**", "**"])
```

📖 **NOTE**

When you modify the whitelist for a workflow asset, the system automatically obtains the information of the algorithm asset that the workflow version depends on and sets the whitelist for the algorithm asset as well.

4.10 Advanced Capabilities

4.10.1 Partial Execution

Workflows support predefined scenarios to enable partial execution. You can split the DAG into different branches based on the scenarios during workflow development. Then, you can run each branch independently as a separate workflow. The sample code is as follows:

```
workflow =wf.Workflow(
    name="image_cls",
    desc="this is a demo workflow",
    steps=[label_step, release_data_step, training_step, model_step, service_step],
    policy=wf.policy.Policy(
        scenes=[
            wf.policy.Scene(
                scene_name="Model training",
                scene_steps=[label_step, release_data_step, training_step]
            ),
            wf.policy.Scene(
                scene_name="Service deployment",
                scene_steps=[model_step, service_step]
            ),
        ]
    )
)
```

This example shows a workflow with five phases. The policy defines two preset scenarios: model training and service deployment. When the workflow is published

to the running state, partial execution is disabled by default and all phases run. You can specify certain scenarios to run on the global configuration page.

> **NOTE**
>
> You can define the same phase in different running scenarios using partial execution. However, you must ensure that the data dependency between phases is correct. Partial execution can only be configured and used in the running state and cannot be debugged in the development state.

4.10.2 Using Big Data Capabilities (DLI/MRS) in a Workflow

Function

This phase calls MRS for big data cluster computing. It is used for batch data processing and model training.

Application Scenarios

You can use MRS Spark for big data computing in this phase.

Examples

On the MRS console, check available MRS clusters of your account. If no MRS cluster is available, create one with Spark selected.

You can use MrsStep to create a job phase. The following is an example of defining a MrsStep:

- **Specifying a boot script and cluster**
  ```
  from modelarts import workflow as wf
  # Define a MrsJobStep using MrsStep.

  algorithm = wf.steps.MrsJobAlgorithm(
      boot_file="obs://spark-sql/wordcount.py",    # OBS path to the boot script
      parameters=[wf.AlgorithmParameters(name="run_args", value="--master,yarn-cluster")]
  )
  inputs = wf.steps.MrsJobInput(name="mrs_input", data=wf.data.OBSPath(obs_path="/spark-sql/
  mrs_input/"))  # OBS path to the input data
  outputs = wf.steps.MrsJobOutput(name="mrs_output",
  obs_config=wf.data.OBSOutputConfig(obs_path="/spark-sql/mrs_output")) # OBS path to the output
  data
  step = wf.steps.MrsJobStep(
          name="mrs_test",          # Step name, which can be customized
          mrs_algorithm=algorithm,
          inputs=inputs,
          outputs=outputs,
          cluster_id="cluster_id_xxx"   # MRS cluster ID
  )
  ```

- **Configuring a cluster and boot script**
  ```
  from modelarts import workflow as wf
  # Define a phase using MrsJobStep.
  run_arg_description = "Program execution parameter, which is used as the program running
  environment parameter. The default value is (--master,yarn-cluster)".
  app_arg_description = "Program execution parameter, which is used as the input parameter of the
  boot script, for example, (--param_a=3,--param_b=4). This parameter is optional and left blank by
  default."
  mrs_outputs_description = "Data output path, which can be obtained from train_url in the parameter
  list."
  cluster_id_description = "cluster id of MapReduce Service"
  ```

```
algorithm = wf.steps.MrsJobAlgorithm(
    boot_file=wf.Placeholder(name="boot_file",
                    description="Program boot script",
                    placeholder_type=wf.PlaceholderType.STR,
                    placeholder_format="obs"),
    parameters=[wf.AlgorithmParameters(name="run_args",
                        value=wf.Placeholder(name="run_args",
                                    description=run_arg_description,
                                    default="--master,yarn-cluster",
                                    placeholder_type=wf.PlaceholderType.STR),
                    ),
            wf.AlgorithmParameters(name="app_args",
                        value=wf.Placeholder(name="app_args",
                                    description=app_arg_description,
                                    default="",
                                    placeholder_type=wf.PlaceholderType.STR)
                    )
        ]
    )

inputs = wf.steps.MrsJobInput(name="data_url",
                    data=wf.data.OBSPlaceholder(name="data_url",object_type="directory"))

outputs = wf.steps.MrsJobOutput(name="train_url",
obs_config=wf.data.OBSOutputConfig(obs_path=wf.Placeholder(name="train_url",
placeholder_type=wf.PlaceholderType.STR,
placeholder_format="obs",description=mrs_outputs_description)))

mrs_job_step = wf.steps.MrsJobStep(
    name="mrs_job_test",
    mrs_algorithm=algorithm,
    inputs=inputs,
    outputs=outputs,
    cluster_id=wf.Placeholder(name="cluster_id", placeholder_type=wf.PlaceholderType.STR,
description=cluster_id_description, placeholder_format="cluster")
    )
```

- **Using an MRS phase on the console**

  After a workflow is published, configure phase parameters such as the data input, data output, boot script, and cluster ID on the workflow configuration page.

4.11 FAQs

4.11.1 How Do I Obtain Training Specifications During Debugging in the Development State?

When debugging and running a workflow in the development state, you need to manually configure the training specifications. For details, see **Obtaining Resources**.

4.11.2 How Do I Implement Multiple Branches?

You can use two modes to implement a multi-branch workflow. For details, see **Condition Phase** and **Branch Control**. The branch control mode is more flexible and recommended.

4.11.3 How Do I Import Objects?

When creating a workflow, required objects are imported through workflow packages. The details are as follows:

```
from modelarts import workflow as wf
```

Import the data package:

```
wf.data.DatasetTypeEnum
wf.data.Dataset
wf.data.DatasetVersionConfig
wf.data.DatasetPlaceholder
wf.data.ServiceInputPlaceholder
wf.data.ServiceData
wf.data.ServiceUpdatePlaceholder
wf.data.DataTypeEnum
wf.data.ModelData
wf.data.GalleryModel
wf.data.OBSPath
wf.data.OBSOutputConfig
wf.data.OBSPlaceholder
wf.data.SWRImage
wf.data.SWRImagePlaceholder
wf.data.Storage
wf.data.InputStorage
wf.data.OutputStorage
wf.data.LabelTask
wf.data.LabelTaskPlaceholder
wf.data.LabelTaskConfig
wf.data.LabelTaskTypeEnum
wf.data.MetricsConfig
wf.data.TripartiteServiceConfig
wf.data.DataConsumptionSelector
```

Import the policy package:

```
wf.policy.Policy
wf.policy.Scene
```

Import the steps package:

```
wf.steps.MetricInfo
wf.steps.Condition
wf.steps.ConditionTypeEnum
wf.steps.ConditionStep
wf.steps.LabelingStep
wf.steps.LabelingInput
wf.steps.LabelingOutput
wf.steps.LabelTaskProperties
wf.steps.ImportDataInfo
wf.steps.DataOriginTypeEnum
wf.steps.DatasetImportStep
wf.steps.DatasetImportInput
wf.steps.DatasetImportOutput
wf.steps.AnnotationFormatConfig
wf.steps.AnnotationFormatParameters
wf.steps.AnnotationFormatEnum
wf.steps.Label
wf.steps.ImportTypeEnum
wf.steps.LabelFormat
wf.steps.LabelTypeEnum
wf.steps.ReleaseDatasetStep
wf.steps.ReleaseDatasetInput
wf.steps.ReleaseDatasetOutput
wf.steps.CreateDatasetStep
wf.steps.CreateDatasetInput
wf.steps.CreateDatasetOutput
wf.steps.DatasetProperties
```

```
wf.steps.SchemaField
wf.steps.ImportConfig
wf.steps.JobStep
wf.steps.JobMetadata
wf.steps.JobSpec
wf.steps.JobResource
wf.steps.JobTypeEnum
wf.steps.JobEngine
wf.steps.JobInput
wf.steps.JobOutput
wf.steps.LogExportPath
wf.steps.MrsJobStep
wf.steps.MrsJobInput
wf.steps.MrsJobOutput
wf.steps.MrsJobAlgorithm
wf.steps.ModelStep
wf.steps.ModelInput
wf.steps.ModelOutput
wf.steps.ModelConfig
wf.steps.Template
wf.steps.TemplateInputs
wf.steps.ServiceStep
wf.steps.ServiceInput
wf.steps.ServiceOutput
wf.steps.ServiceConfig
wf.steps.StepPolicy
```

Import the workflow package:

```
wf.Workflow
wf.Subgraph
wf.Placeholder
wf.PlaceholderType
wf.AlgorithmParameters
wf.BaseAlgorithm
wf.Algorithm
wf.AIGalleryAlgorithm
wf.resource
wf.SystemEnv
wf.add_whitelist_users
wf.delete_whitelist_users
```

4.11.4 How Do I Locate Running Errors?

If a workflow reports an error when it runs in run mode, follow these steps:

1. Check if you have the latest version of the SDK package installed to avoid package version inconsistency.

2. Check if your SDK code follows the specifications. See the corresponding code example for details.

3. Check if the content you entered during the execution is correct and matches the format required in the prompt message.

4. Find the code line where the error occurs based on the error information and analyze the context logic.

Common Errors With Historical SDK Packages

- An error is reported when the service deployment phase is running.

```
'weight': 100, 'specification':'custom',
'cluster_id':'******','custom_spec':{'cpu':2.5, 'memory':
1024},'envs':{'******': '******', '******': '******'}}]"

Note that:(1) The "[" at the beginning and "]" at the end
are required.
            (2) The sum of the weights must be equal to 100.
            (3) All model must have the same model name. Two
model versions cannot be the same.
[{'model_name':'        :','model_version':'       ','specific
ation':'                ','weight':100}]
Traceback (most recent call last):
```

- The following execution error information appears after you configure the service parameters:

```
\placeholder.py , tine 50, in __eq__
        return self.name == obj.name and \

AttributeError: 'str' object has no attribute 'name'
```

Solutions

The two common errors can be rectified by upgrading the SDK to the latest version.