Cisco Crosswork Workflow Manager 2.1 Workflow Creator Guide

First Published: 2026-01-29

Last Modified: 2026-01-29

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CHAPTER 1

Feature overview

This section contains the following topics:

- Overview, on page 1
- Workflow definition features, on page 1
- States, on page 6
- Operation state overview, on page 9
- Switch state overview, on page 11
- Sleep state, on page 12
- Inject state, on page 13
- ForEach state, on page 13
- Parallel state, on page 15
- Callback state, on page 15
- State data, on page 16

Overview

Workflows help you automate business processes in a standardized manner, bridging the gap between expressing and modelling business logic.

Workflow definitions are written based on the Serverless Workflow pecification. For CWM, only a subset of the full specification is supported. This chapter describes all the supported features and gives practical examples for many of them.

Workflow definition features

A new workflow can be defined in JSON format. The structure of the workflow definition is described in the Serverless Workflow specification.

The supported high-level components are as follows:

- id
- name
- description
- version

- start
- retries
- errors
- functions
- states
- metadata

Toplevel fields

Table 1: Toplevel fields

Parameter	Description
id	Unique identifier for the workflow.
name	Workflow name.
version	Workflow version based on Semantic Versioning.
specVersion	Version of the Serverless Workflow specification release this definition adheres to. The current CWM implementation corresponds to the 0.9 specification.
description	Workflow description text.
start	State to be executed first.

Example:

```
{
"id": "MyWorkflow",
"version": "1.0.0",
"specVersion": "0.9",
"name": "My Workflow",
"description": "My Workflow Description",
"start": "SomeState",
"states": [],
"functions": [],
"retries":[]
}
```

Retry definitions

Retry definitions are policies that can be assigned to workflow activities to control how the workflow engine deals with retries in the event of failure.

The following properties of retry definitions are supported.

Table 2: Retry definitions

Parameter	Definition
name	Definition name.
delay	Time delay between retry attempts in ISO 8601 format, for example "PT30S" for a 30 second delay.
maxAttempts	Maximum number of attempts. Set to 0 for infinite retries. For no retries, set to 1.
maxDelay	Maximum amount of delay between retry attempts. Uses ISO 8601 format.
multiplier	Used to multiply delay value, if provided before each retry attempt. This is a float value. For example, if the initial delay is 30 seconds, and the multiplier is 1.5, the retries will increase by 50% each time.

Example:

```
"retries": [
     {
         "name": "Default",
         "delay": "PT1M",
         "maxAttempts": 5,
         "multiplier": 1.2
         "maxDelay": "PT3M"
     }
 ]
```

Error definitions

Error definitions describe errors that can occur during workflow execution. Whilst the serverless specification supports referencing an external file (JSON) that lists the errors, CWM will only handle errors defined in the Workflow definition.

The following properties of error definitions are supported.

Table 3: Error definitions

Parameter	Definition
name	Definition name.
code	Error code that could be returned. Currently, this field is not used for error matching.
description	Should describe the error message. This description is used to match against the error returned by activities.

Note The Serverless Workflow specification doesn't have an option to specify an error message. This means that currently the `description` is being used for matching against errors.

Example:

```
"errors": [
    {
        "name": "My Custom Error",
        "code": 0,
        "description": "Specific Error Message"
    }
    ]
```

Function definitions

Function definitions describe the functions available for the workflow to execute and the name of the adapter and activity that should be invoked by the engine when that function is invoked. While the Serverless Workflow specification supports various types of functions, CWM supports only those custom type functions that map to activities exposed via Adapters.

The following properties of function definitions are supported.

Table 4: Function definitions

Parameter	Definition
name	Name of function definition.
operation	Defines the adapter name and activity name that should be invoked by the engine. Format is <adapter name>.<activity name>. For example: The NSO Adapter has an activity called `RestconfGet`. An operation for this would be the name of the activity as registered in the worker, such as `RestconfGet`. Note that this name is case-sensitive.
metadata	Allows modelling of information beyond the core definition of the Serverless Workflow specification. The "worker" key is used to define which Taskqueue the activities will be executed on. CWM supports the concept of Workers that execute an Activity and are assigned Taskqueues that they listen to. To schedule an activity to run, the workflow engine places the activity on a Taskqueue. A worker process picks up the tasks to execute from the Taskqueue and executes the activity.

Example:

```
"functions": [
        {
            "name": "NSO.RestconfGet",
            "operation": "restconf_Get"
            "metadata":  {
                "worker": "defaultWorker"
            }
        },
        {
            "name": "NSO.RestconfPut",
            "operation": "restconf_Put"
            "metadata":  {
                "worker": "defaultWorker"
            }
        },
{
    "name": "NSO.RestconfPost",
    "operation": "restconf_Post"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.RestconfPatch",
    "operation": "restconf_Patch"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.RestconfDelete",
    "operation": "restconf_Delete"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "NSO.SyncFrom",
    "operation": "device_SyncFrom"
    "metadata":  {
        "worker": "defaultWorker"
    }
},
{
    "name": "REST.Post",
    "operation": "rest_Post"
    "metadata":  {
        "worker": "defaultWorker"
    }
} ]
```

SubFlowRef definitions

SubFlowRef definitions are used for invoking child workflows within a parent workflow. With child workflows, you can:

- Separate the parent workflow code and workers from the child workflow code and workers.
- Split the workload done by the workflow into smaller chunks for better separation of event history. This is especially helpful when your workflow is intended to spawn large numbers of activity executions.

The following properties of `subFlowRef` definition are supported:

Table 5: SubFlowRef Properties

Parameter	Description
workflowId	Child workflow unique id.
version	Child workflow version.
invoke	Specifies if the child workflow should be invoked `sync` or `async`. Default is `sync`, which means workflow execution should wait until the child workflow completes.

Parameter	Description
onParentComplete	If invoke is `async`, specifies if child workflow execution should terminate or continue when parent workflow completes. Default is `terminate`.

Example:

```
"states": [
    {
    "end": true,
    "name": "SpawnChildWorkflow",
    "type": "operation",
    "actions": [
        {
        "subFlowRef": {
            "version": "1.0",
            "workflowId": "subtest",
            "invoke": "sync",
            "onParentComplete": "terminate"
        }
      }
    ]
  }
]
```

States

States define the building blocks of workflow execution logic. Different types of states provide control flow logic to the Execution Engine and also allow you to define which activities to execute.

Common state properties

The following properties are common to all states.

For any given state, you can only have one `transition` or `end` object. At least one must be present.

Table 6: Common state properties

Parameter	Definition
name	State name.
type	Supported types are: "operation", "switch", "sleep", "inject", "foreach".
transition	Next transition of workflow - see below for further details. *Not applicable to SwitchState. For switch state, the transition option is defined on a per condition basis.*
end	If the workflow should end after this state - see below for further details. *Not applicable to SwitchState. For switch state, the end option is defined on a per condition basis.*

Parameter	Definition
stateDataFilter	Filter data input and output for the state - not applicable to "sleep" state.
onErrors	Defines error handling for a given state, see below for further details. Can match based on Error Definition and control transition/end based on matched error including Compensation.
usedForCompensation	If `true`, this state is used to compensate another state. Default: `false`.
compensatedBy	Unique name of state which is responsible for compensation of this state. State identified here, is executed if "compensate" is set to true for transition/end property.

Compensation

Compensation lets you define ways to undo the work done as part of a workflow. For each state, you can define a compensation state. If, during execution, a condition is reached where compensation logic should be executed, a `compensate` flag can be set when defining a `transition` or `end`. The flag will result in executing states that are to be `usedForCompensation`. Refer to the Workflow Serverless specification for more information: Workflow compensation.

In CWM, each state marked for compensation is added to a LIFO (Last In First Out) queue.

Transition

The Serverless Workflow specification supports defining `transition` either as a `string` or as an `object` with further properties. The current CWM implementation supports the `object` format and the `nextState` and `compensate` properties only.

Table 7: Transition

Parameter	Definition
nextState	The name of state that workflow will transition to next.
compensate	If set to `true`, triggers workflow compensation before next transition is taken. Default: `false`.

End

The Serverless Workflow specification supports defining `end` either as a `string` or as an `object` with further properties. The current CWM implementation supports the `object` format and the `nextState` property only.

Table 8: End states

Parameter	Definition
terminate	Boolean value to define if this state should terminate the workflow.
compensate	If set to true, triggers workflow compensation before execution completes. Default: `false`.

stateDataFilter

State Data Filters allow you to define input and output data filters. Input Data filters allow you to select data that is required. Output Data filters are applied before transitioning to the next state, allowing you to filter data to be passed into the next state. More information on State Data Filters can be found in the Serverless Workflow specification. Both the input and output filters are workflow expressions defined in jq. If no filters are specified, then all data is passed.

Table 9: stateDataFilter

Parameter	Definition
input	Input filter jq expression.
output	Output filter jq expression.

Example:

```
"states": [
        {
            "name": "step1",
            "type": "operation",
            "stateDataFilter" : {
                "input": "${ . }"
                "output": "${ . }"
            }
            "transition": {
                "nextState": "downloadImage"
            }
        },
        {
            "name": "step2",
            "type": "operation",
            "end": {
                "terminate": "true"
            }
        }
    ]
```

onErrors

The onErrors property for a state defines errors that may occur during state execution and how they should be handled. You can find more information about onErrors in the Serverless Workflow documentation.

Table 10: onErrors

Parameter	Definition
errorRef or errorRefs	Define either a single errorDef or array of ErrorDefs to match for this state.
transition	Next transition of workflow if the error returned in state matches any of the error description in errorRef/errorRefs. Only `transition` or `end` can be defined.

Parameter	Definition
end	The workflow should end if the error returned in state matches any of the error description in errorRef/errorRefs. Only `transition` or `end` can be defined.

Example:

```
"onErrors": [
        {
            "errorRef": "My Custom Error",
            "end" : {
                "terminate": true
                "compensate": true
            }
        }
    ]
```

Operation state overview

The serverless workflow specification permits operation states to define sets of actions to be executed in sequence or parallel. CWM supports execution of actions in sequence only.

An specification defines invocation of three different types of services:

- Execution of function definition. This is the only type of service that CWM currently supports.
- Execution of another workflow definition as a child workflow. This is not supported in the current implementation.
- Referencing events that may be "produced" or "consumed". This is not supported in the current implementation.

Action

Action definition specifies the function that should be executed for this state. The following properties are supported:

Parameter	Description
name	Action name.
functionRef	Object which defines the name of the function to be executed, and optionally arguments to pass into the activity the function points to. See below for further details.
retryRef	Name of retry definition defined globally. For example, `default`.
sleep	Object that optionally defines time to sleep either before or after action execution. See below for further details.
actionDataFilter	Filter to control what data should be passed to action, how to filter the results returned by action, and where to store the filtered results in the global state data. See below for further details.

functionRef

Parameter	Description
refName	Name of function referencing the function definition.

arguments	Arguments to be passed to the function. This can be a JSON object with complex structure. For Adapter activities, the structure has to be JSON, as follows:
	`{`
	`"input": {`
	`...`
	`},`
	`"resource": {`
	`...`
	`}`
	`}`

actionDataFilter

For detailed information on actionDataFilter with examples, see see this Serverless Workflow specification section.

Parameter	Description
fromStateData	Workflow expression in jq that filters data from state data to pass into function.
useResults	Boolean flag to control whether data returned from function execution should added/merged into state data output.
results	Workflow expression in jq that filters the data returned from function execution. Ignored if useResults is `false`. Default: `true`.
toStateData	Workflow expression defines state data where the results should be added/merged. If not specified, results merged at top level.

sleep

Sleep specifies the amount of time to to pause before or after executing a workflow function.

Parameter	Description
before	Amount of time to sleep before function is executed in ISO 8601 format e.g. "PT30S" - sleep for 30 seconds.
after	Amount of time to sleep after function is executed in ISO 8601 format e.g. "PT30S" - sleep for 30 seconds.

```
{
    "id": "example",
    "version": "1.0",
    "specVersion": "0.9",
    "start": "step1",
    "functions": [
        {
            "name": "NSO.RestconfPost",
            "operation": "RestconfPost"
            }
        ],
        "retries": [
            {
                "name": "Default",
                "maxAttempts": 5,
                "delay": "PT30S",
                "multiplier": 1.1
            }
        ],
        "states": [
            {
                "name": "step1",
                "type": "operation",
                "sleep": {
                    "before": "PT1M"
                },
                "actions": [
                    {
                        "retryRef": "Default",
                        "name": "showVersion",
                        "functionRef": {
                            "refName": "NSO.RestconfPost",
                            "arguments": {
                                "input": {
                                "path": "restconf/operations/devices/device=${ .deviceName
 }/live-status/tailf-ned-cisco-ios-stats:exec/any",
                                    "data": "{\"input\": {\"args\": \"show version\"}}"
                                }
                            }
                        },
                        "actionDataFilter": {
                            "results": "${ if (.data) then .data |
fromjson.\"tailf-ned-cisco-ios-stats:output\".result else null end }",
                            "toStateData": "${ .showVersionPreCheck }"
                        }
                    }
                ],
                "end": {
                    "terminate": "true"
                }
            }
        ]
    }
```

Switch state overview

Switch states enable you to define decision points to route the workflow to a given path based on certain conditions. The Serverless Workflow specification supports both data-based conditions and event-based conditions. CWM supports data-based conditions only.

dataConditions

The data condition property of Switch state is an array of conditions that are evaluated by the Execution engine. The Execution engine will select the first condition it matches and proceed along that path. If there are subsequent conditions that also match, they will be ignored.

Parameter	Description
name	Condition name.
condition	Workflow expression in jq that represents the condition. Must evaluate to `true`/`false`.
transition	Next transition of workflow if the condition matches.
end	The workflow should end if the condition matches.

You can provide only the `transition` object or the `end` object. At least one must be present.

defaultCondition

The default condition that is applied if none of the conditions match.

Parameter	Description
transition	Next transition of workflow if no conditions are matched.
end	The workflow should end if condition matches.

You can provide only the `transition` object or the `end` object. At least one must be present.

```
{
    "name": "ConditionName",
    "type": "switch",
    "dataConditions": [
        {
            "name": "IsTrue",
            "condition": "${ true  }",
            "transition": {
                "nextState": "TrueState"
            }
        },
        {
            "name": "IsFalse",
            "condition": "${ false }",
            "transition": {
                "nextState": "FalseState"
            }
        }
    ],
    "defaultCondition": {
        "end": {
            "terminate": true
        }
    }
}
```

Sleep state

Sleep state pauses workflow execution for a given duration.

Parameter	Description
duration	Duration the workflow should sleep for in ISO8601 format. For example, PT1M results in workflow sleeping for 1 minute.

```
{
    "name": "Sleep3Minutes",
    "type": "sleep",
    "duration": "PT3M",
    "transition": {
        "nextState": "NextState"
    }
}
```

Inject state

Use Inject state to inject static data into the State Data.

Parameter	Description
data	JSON object added to State Data.

```
{
    "id": "example",
    "version": "1.0",
    "specVersion": "0.9",
    "start": "HelloWorld",
    "states": [
        {
            "name": "HelloWorld",
            "type": "inject",
            "data": {
                "name": "Cisco",
                "message": "Hello World"
            },
            "stateDataFilter":{
                "output": "${ .message + \" from \" + .name + \"!\"  }"
            },
            "end": {
                "terminate": "true"
            }
        }
    ]
}
```

ForEach state

ForEach state allows you to define a set of actions to execute for each element in an array or list defined in State Data. For example, for each device in device array, check that the devices are in sync. While the serverless workflow specification defines support for Parallel and Sequential execution of actions, current implementation only supports sequential execution of actions for each element in array.

Parameter	Description
inputCollection	Workflow expression in jq that points to an array in State Data.

Parameter	Description
iterationParam	Name of the parameter that can be referenced in action for each data element.
outputCollection	Workflow expression in jq that points to an array in State Data that the result will be appended to. If array doesn't exist, it will be created.

```
{
    "id": "example",
    "version": "1.0",
    "specVersion": "0.9",
    "start": "InjectData",
    "functions": [
        {
            "name": "HelloWorld",
            "operation": "HelloWorld"
        }
    ],
    "states": [
        {
            "name": "InjectData",
            "type": "inject",
            "data": {
                "people": [
                    {
                        "Firstname": "Peter",
                        "Surname": "Parker"
                    },
                    {
                        "Firstname": "Thor",
                        "Surname": "Odinson"
                    },
                    {
                        "Firstname": "Bruce",
                        "Surname": "Banner"
                    }
                ]
            },
            "transition":{
                "nextStat": "SayHelloToEveryone"
            }
        },
        {
            "name": "SayHelloToEveryone",
            "type": "foreach",
            "inputCollection": "${ .people }",
            "iterationParam": "person",
            "outputCollection": "${  .messages }",
            "actions": [
                {
                    "name": "SayHello",
                    "functionRef":{
                        "refName": "HelloWorld",
                        "arguments": {
                            "name": "${ .person.Firstname + \" \" + .person.Surname }"
                        }
                    }
                }
            ],
            "end": {
                "terminate": "true"
```

```
                    }
                }
            ]
        }
```

Parallel state

Parallel state allows you to define a collection of branches that are executed in parallel. Each branch in a state can define its own set of actions. Once the execution has completed, the parallel branches are joined into current path based on the `completionType` attribute.

The `completionType` attribute can define two values:

- `allOf`: All branches must complete execution before state can transition/end. This is the default value.
- `atLeast`: State can transition/end if the number of branches specified in `atLeast` has completed execution. If `completionType` attribute is `"atLeast"`, `numCompleted` must also be set.

Parameter	**Description**
completionType	Define how to evaluate completion of state based on branch execution. `"allOf"` or `"atLeast"`. Default: `"allOf"`.
numCompleted	If `completionType` is `"atLeast"`, this value must be specified. Defines the minimum number of branches that must be completed for the execution to proceed.

Branches

Following is a list of branches that are to be executed in parallel state. For more information on branches, see the Serverless Workflow Specification documention on the Parallel State Branch, https://github.com/serverlessworkflow/specification/blob/0.9.x/specification.md#Parallel-State-Branch.

Parameter	**Description**
name	Name of branch.
actions	Actions to execute for this branch. A branch can support an array of actions. The definition for each action is the same as for Operation state type.

Callback state

Callback state allows workflow designers to introduce manual tasks (human intervention points) into their workflows. Within **Callback**, the **action** property defines a function call that triggers an external activity/service (note that stating the function call is required for this state). Once the action executes, the callback state waits for a CloudEvent (defined via the `eventRef`property), which indicates the completion of the manual decision by the called service.

Parameter	Description	Type	Required
name	Unique State name. Must follow the Serverless Workflow Naming Convention	string	yes
type	State type	string	yes
action	Defines the action to be executed	object	yes
eventRef	References a unique callback event (Form ID) in the defined workflow events	string	yes
transition	Next transition of the workflow after callback event has been received	string or object	yes (if end is not defined)
end	Is this state an end state	boolean or object	yes (if transition is not defined)

Note According to the Serverless workflow spec, you need to include the `action` parameter for the callback state, although it is not required for triggering the task itself (the *callback event*).

State data

State data plays an important role during the lifecycle of the workflow. A state can filter data, inject data, and add data. Jq plays an important role in data filtering, creation and manipulation. For more information on how data can be handled, see the Serverless Workflow specification.

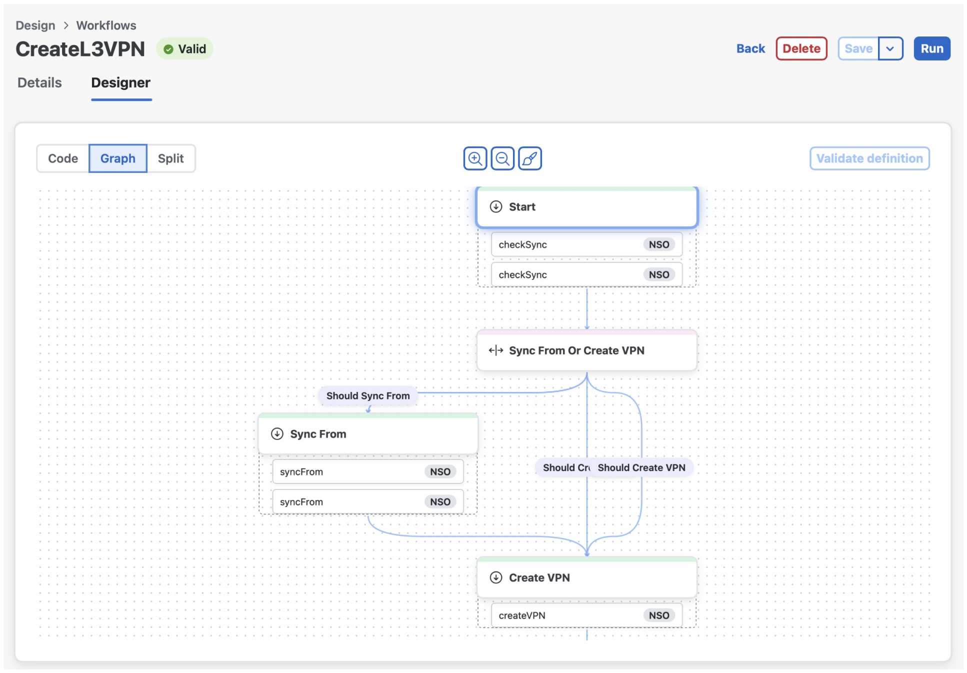
When creating workflows in CWM, the following data management rules apply:

- Initial data passed into workflow execution is passed into State data as input.
- Data output from the last executed state is workflow output.
- If no State Input Filter is specified, all the data is passed into the state.
- If no State Output Filter is specified, all the data is passed into the next state.
- Workflow expressions in jq allow you to filter and manipulate data.
- Actions also allow for filtering data and also, if return data from action should be merged back into state data.
- Filters must return JSON objects. If a jq workflow expression results in a string literal, this will result in an error.
- When working with jq, it is highly recommended to use https://jqplay.org/ to test the jq expressions. Alternatively, you can download jq locally and use it for testing.

Visualize workflow logic

In CWM, choose **Design** > **Workflows**, click on a **workflow name**, then click the **Designer** tab to see a graphical representation of a created workflow.

The **Designer** view lets you trace the workflow's task sequences, decision points, and dependencies, helping you to ensure that the workflow is correctly structured.



CHAPTER 2

Workflow Designer

This section contains the following topics:

- Workflow Designer, on page 19

Workflow Designer

Overview

The Workflow Designer is a low-code visual environment for building and managing workflows using drag-and-drop interactions. It provides a graphical representation of workflows where states, activities, and subflows are arranged within a directed graph. The Designer supports direct manipulation of workflow elements to define sequencing, nesting, and error-handling. Workflow definitions are created and modified through drag-and-drop operations, context menu actions, and keyboard interactions, reducing the need for manual configuration.

The Designer lets you:

- add states and activities,
- define how they connect,
- control sequencing and branching,
- and adjust layout so the graph is easy to understand.

The canvas shows a background grid and uses alignment aids to help you arrange elements cleanly. You can reposition and reorder elements at any time to refine the visual workflow without changing execution logic.

Toolbox

The Workflow Designer toolbox provides the set of elements that can be used to build workflows. Toolbox items are dragged onto the design canvas and dropped into valid locations in the workflow graph.

At a high level, the toolbox includes:

States: used to define workflow structure and execution flow. The list of states supported includes eight types:

- Callback

- Event
- Foreach
- Inject
- Operation
- Parallel
- Sleep
- Switch

Activities: adapter activities (workflow actions) that run within a state. The set of available activities is populated dynamically based on the adapters installed in CWM.

Workflows (Subworkflows): workflows that can be embedded within a state. Available workflows are those already defined in CWM and can be used to modularize and reuse workflow logic.

Events: predefined event types that can be used to trigger or react to specific conditions within a workflow. The available event types are provided by the system and depend on the configured event sources.

Properties drawer

The Workflow Designer provides a context-sensitive properties drawer on the right-hand side of the screen. This drawer is used to configure both the workflow itself and individual workflow elements, and its contents change based on what is currently selected in the canvas.

When the workflow is selected, the drawer displays global workflow settings. These settings align with the Serverless Workflow specification (v0.8) and are organized into tabs such as data handling, timeouts, errors, retries, and other workflow-level options.

When a state is selected, the drawer updates to show configuration options specific to that state type. The available tabs and fields depend on the selected state and correspond to the properties defined for that state in the Serverless Workflow specification.

When an activity or subflow is selected, the drawer presents a different set of options relevant to that element. For example, activity configuration may include parameters such as state data mapping or form selection for human intervention tasks.

All fields in the drawer include inline descriptions and helper text, allowing users to understand and configure options directly in the UI without needing to refer to external documentation.

Visual aids and layout

The designer’s grid and alignment helpers let you:

- visually align elements,
- maintain a clean layout,
- and clearly follow how workflow logic flows from one state to the next.

The canvas also supports zooming and navigation to make working with large or complex workflows easier:

- You can zoom in and out of the workflow graph to focus on details or view the overall structure.
- On macOS, zooming can be performed using a two-finger pinch gesture on the touchpad.

- A center / reset view control is available to quickly return to the default view of the workflow, centering the graph (including the start state) in the canvas.

These tools help users stay oriented and keep workflows readable as they grow.

Designer views

The Workflow Designer provides three view modes, accessible from the tabs in the top-left corner:

- Graph: Displays the visual workflow graph and is used for drag-and-drop design and layout.
- Code: Displays the code of the workflow definition, allowing direct editing.
- Split: Displays the graph and code views side by side, enabling users to see visual changes reflected in the workflow code and vice versa.

These views allow users to work visually, directly in code, or with both representations at the same time, depending on their preference or use case.

CHAPTER 3

Create a workflow tutorial

This section contains the following topics:

• Create a workflow – tutorial, on page 23

Create a workflow – tutorial

This chapter shows you how you can structure a workflow based on a simple example that uses the `operation` and `switch` states to create a VPN service in Cisco NSO for some simulated devices. We go through the example workflow definition part by part to give you an idea how you can use different definition components in creating your original workflows.

If you need full information on how workflows can be defined, refer to the Serverless Workflow specification.

Example workflow overview

The goal of this example workflow is to automatically create a VPN service for Cisco NSO devices.

First, we point to the devices in the data input and then try to perform the NSO `check-sync` operation on them. Then, depending on the result:

• If a device is not in sync with NSO, we push the device to perform a `sync-from`, and only then try to create a VPN for it;

• If it is in sync, we don't perform `sync-from` but directly create a VPN for the device.

If all the steps are executed successfully, CWM reports workflow execution completion and diplays the final data input. The results are visible in Cisco NSO, too. If the execution engine encounters errors while performing a step, it uses the specified `retry` policy. If errors persist beyond the retry limits, the execution engine ends the execution with a **Failed** status.

Go through the sections below to understand how data input, functions, states, actions, and data filters are defined.

Provide data input

The workflow definition usually includes some input data at the beginning of the JSON file. While the provided data is not part of the workflow, it is referred to within the workflow definition and can also be updated between states, if such a data update is defined. For more details, see Workflow data input in the Serverless Workflow Specification.

In this example, we'll only need to provide two user-defined `deviceName` JSON object keys and values, which are the names of the test devices in the local NSO instance, and the `nsoResource` key, where we specify which CWM resource we will be using in the workflow. The workflow data input in JSON should look like this:

```
{
"device0Name": "ce0",
"device1Name": "ce1",
"nsoResource": "NSOLocal"
}
```

Define top-level parameters and functions

A workflow definition starts with the required workflow `id` key. Among other keys, `specVersion` is also required, defining the Serverless Workflow specification release version. The `start` key defines the name of the workflow starting state, but it is not required.

In the `functions` key, you pass in Cisco NSO adapter activity name as function `name`, adapter activity ID as function `operation`, and provide the worker name under `metadata`: `worker` key:

```
{
    "id": "CreateL3VPN",
    "name": "Create Layer3 VPN",
    "start": "start",
    "version": "1.0",
    "functions": [
    {
      "name": "NSO.RestconfPost",
      "metadata": {
        "worker": "cisco.nso.v1.0.1"
      },
      "operation": "cisco.nso.v1.0.1.restconf.Post"
    }
  ],
  "description": "Create an L3 VPN for MPLS devices",
  "specVersion": "0.9"
}
```

Note Effectively, what you do under `functions` is you provide the workflow with the IDs of any activities as they are defined in the Cisco NSO adapter and presented in its `main.go` file. Also, under `metadata` you provide the name of the worker that will execute any actions that refer to the defined function.

Specify retry policy

With the `retries` key, you define the retry policies for state actions in the event that an action fails. Multiple retry policies can be specified under this key and they are reusable across multiple defined state actions.

```
"retries": [
    {
      "name": "Default",
      "maxAttempts": 4,
      "delay": "PT5S",
      "multiplier": 2
    },
            {
              "name": "Custom",
              "maxAttempts": 2,
              "delay": "PT30S",
              "multiplier": 1
            }
          ],
```

Note As you can see, the `Default` policy assumes that a failed action will be retried up to 4 times with an increasing delay between attempts: 5, 10, 20, 40 seconds between consecutive retries.

Define states

Workflow states are the building blocks of a workflow definition. In the present quickstart example, we will be using the `operation` and `switch` states, but others are possible. You can check them in detail in the Workflow states section of the Serverless specification.

Operation state

```
"states": [
        {
        "name": "start",
        "type": "operation",
        "stateDataFilter": {
            "input": "${ . }"
        },
        "actions": [],
        "transition": {
            "nextState": "syncFromOrCreateVPN"
            }
        }
    ]
```

Inside the `operation` state, apart from state `name` and `type`, you define:

- `stateDataFilter`: Point to the data input defined at the beginning of the example JSON file. In the `input` parameter, we state `${ . }`, which is a jq expression that means: "use the whole of the data input existing at this point of workflow execution".

Note For more information on how jq expressions are used in workflows, see the Workflow expressions chapter in the Serverless Workflow specification.

- `actions`: Specify the function to be used by the action, and two basic `arguments`: `input` and `config`. Read more in the subsection below.

- `transition` or `end`: Point to the next state to which the workflow should transition after executing the present one. If there are no more steps to be executed, use `end`.

Switch state

```
      {
        "name": "syncFromOrCreateVPN",
        "type": "switch",
        "dataConditions": [
           {
             "name": "shouldSyncFrom",
            "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 != \"in-sync\"
else null end }",
             "transition": {
               "nextState": "syncFrom"
             }
           },
           {
             "name": "shouldCreateVPN",
            "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 == \"in-sync\"
else null end }",
            "transition": {
               "nextState": "createVPN"
             }
          }
         ]
      }
```

Inside the `switch` state, apart from state `name` and `type`, you define:

- `dataConditions`: Define the conditions to be met by a device to be transitioned to a specified next state. You can view the `switch` state as a "gateway" for the workflow, which directs the devices to appropriate states based on their status. Using the jq expression `${ if (.checkSyncResult0) then .checkSyncResult0 == \"in-sync\" else null end }` in the `condition` parameter, we create a boolean value that, if it evaluates to `true`, is used to transition the device directly to the `CreateVPN` state.

Specify actions

Let's analyse `actions` on the basis of the `checkSync` action of the `operation` state for device `ce0`.

```
{
       "name": "checkSync",
       "retryRef": "Default",
       "functionRef": {
         "refName": "NSO.RestconfPost",
         "arguments": {
           "input": {
             "path": "restconf/operations/tailf-ncs:devices/device=${ .device0Name
}/check-sync"
           },
           "config": {
             "resourceId": "${ .nsoResource }"
           }
         }
       },
       "actionDataFilter": {
         "results": "${ if (.data) then .data | .\"tailf-ncs:output\".result else null end
 }",
         "toStateData": "${ .checkSyncResult0 }"
       }
     }
```

Among the possible parameters, two are especially useful to consider:

- `functionRef`: Refer to the function (aka an activity, from the NSO adapter perspective) to be used in action execution. Here, you need to pass in some `arguments`:
  - `input`:

        - `path`: Point to a path for the adapter to send the request to.
        - `data`: Forward any data to be included in the request (not applicable for the `checkSync` action).

    - `config`:

        - `resourceId`: Provide the ID of the resource you created for an external service. In the example workflow, the local host and the default port of the Cisco NSO instance is provided. The resource also points to the secret ID, which is used to provide authentication data for an external service. In this case, that will be the `username` and `password` to the Cisco NSO instance.

- `actionDataFilter`: Define how to process the data passed on in the `checkSync` response from NSO:

    - `results`: Use the jq expression `"${ if (.data) then .data | .\"tailf-ncs:output\".result else null end }"` to handle incoming NSO data. Using the `.result` you cherrypick the `result` key value. In this case (if the device is in the in-sync state), the output of the expression would be `"in-sync"`.
    - `toStateData`: Take the output of the expression defined in the `results` parameter above and save it as a key and value pair inside the workflow input data under any name that you pick. In this case, `.checkSyncResult0`.

Example workflow definition

The following example workflow definition is the end result of the workflow creation process presented in this chapter.

For a complete procedure on how to execute the example workflow in CWM and get tangible results in Cisco NSO, see the CWM Getting Started guide.

```
{
  "id": "CreateL3VPN-1.0",
  "name": "CreateL3VPN",
  "start": "start",
  "states": [
    {
      "name": "start",
      "type": "operation",
      "actions": [
        {
          "name": "checkSync",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device0Name
}/check-sync"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | .\"tailf-ncs:output\".result else null
end }",
              "toStateData": "${ .checkSyncResult0 }"
            }
          },
          {
            "name": "checkSync",
            "retryRef": "Default",
            "functionRef": {
              "refName": "NSO.RestconfPost",
              "arguments": {
                "input": {
                  "path": "restconf/operations/tailf-ncs:devices/device=${ .device1Name
}/check-sync"
                },
                "config": {
                  "resourceId": "${ .nsoResource }"
                }
              }
            },
            "actionDataFilter": {
              "results": "${ if (.data) then .data | .\"tailf-ncs:output\".result else null
end }",
              "toStateData": "${ .checkSyncResult1 }"
            }
          }
        ],
        "transition": {
          "nextState": "syncFromOrCreateVPN"
        },
        "stateDataFilter": {
          "input": "${ . }"
        }
      },
      {
        "name": "syncFromOrCreateVPN",
        "type": "switch",
        "dataConditions": [
          {
            "name": "shouldSyncFrom",
            "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 != \"in-sync\"
else null end }",
            "transition": {
              "nextState": "syncFrom"
            }
          },
          {
            "name": "shouldCreateVPN",
            "condition": "${ if (.checkSyncResult0) then  .checkSyncResult0 == \"in-sync\"
else null end }",
            "transition": {
              "nextState": "createVPN"
            }
          },
          {
            "name": "shouldSyncFrom",
            "condition": "${ if (.checkSyncResult1) then  .checkSyncResult1 != \"in-sync\"
else null end }",
            "transition": {
              "nextState": "syncFrom"
            }
          },
          {
            "name": "shouldCreateVPN",
            "condition": "${ if (.checkSyncResult1) then  .checkSyncResult1 == \"in-sync\"
else null end }",
          "transition": {
            "nextState": "createVPN"
          }
        }
      ],
      "defaultCondition": {
        "end": {
          "terminate": true
        }
      }
    },
    {
      "name": "syncFrom",
      "type": "operation",
      "actions": [
        {
          "name": "syncFrom",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device0Name
}/sync-from"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | .\"tailf-ncs:output\".result else null
end }",
            "toStateData": "${ .syncFromResult0 }"
          }
        },
        {
          "name": "syncFrom",
          "retryRef": "Default",
          "functionRef": {
            "refName": "NSO.RestconfPost",
            "arguments": {
              "input": {
                "path": "restconf/operations/tailf-ncs:devices/device=${ .device1Name
}/sync-from"
              },
              "config": {
                "resourceId": "${ .nsoResource }"
              }
            }
          },
          "actionDataFilter": {
            "results": "${ if (.data) then .data | .\"tailf-ncs:output\".result else null
end }",
            "toStateData": "${ .syncFromResult1 }"
          }
        }
      ],
      "transition": {
        "nextState": "createVPN"
      }
    },
    {
      "end": {
          "terminate": true
        },
        "name": "createVPN",
        "type": "operation",
        "actions": [
          {
            "name": "createVPN",
            "retryRef": "Custom",
            "functionRef": {
              "refName": "NSO.RestconfPost",
              "arguments": {
                "input": {
                  "data":
"[illegible]
 ",
                  "path": "restconf/data/l3vpn:vpn"
                },
                "config": {
                  "resourceId": "${ .nsoResource }"
                }
              }
            },
            "actionDataFilter": {
              "results": "${ if (.status) then .status else null end }",
              "toStateData": "${ .createServiceResult }"
            }
          }
        ]
      }
    ],
    "retries": [
      {
        "name": "Default",
        "delay": "PT30S",
        "multiplier": 2,
        "maxAttempts": 4
      },
      {
        "name": "Custom",
        "delay": "PT10S",
        "multiplier": 1,
        "maxAttempts": 2
      }
    ],
    "version": "1.0",
    "functions": [
      {
        "name": "NSO.RestconfPost",
        "metadata": {
          "worker": "cisco.nso.v1.0.3"
        },
        "operation": "cisco.nso.v1.0.3.restconf.Post"
      }
    ],
    "description": "",
    "specVersion": "0.9"
}
```

CHAPTER 4

Utilities

This section contains the following topics:

- System-defined utilities, on page 31
- Expression functions, on page 33
- Input schema validation, on page 34

System-defined utilities

System-defined utilities are functions that expose activities to be used by a workflow creator. Invoking them as actions inside a workflow helps fulfill basic tasks without the need to create custom adapters. They come pre-packaged and are ready-to-use in any workflow definition.

Invoke a system utility in a workflow

To use a system utility function in a workflow execution, you must first define it under the `functions` key in the workflow. Let's take the `NoOp` function as an example.

```
"functions": [
  {
    "name": "noop",
    "metadata": {
      "worker": "default"
    },
    "operation": "system.function.@latest.common.NoOp"
  }
],
```

- `name`: A user-defined name for the utility function referred to inside the workflow definition.
- `operation`: This is where you provide the utility function name (name of activity as with an adapter activity).
- `worker`: Defining a worker is mandatory for all utility functions. Use the `default` worker for this.

Make sure to define the `worker` key with the value `default` for each utility function.

```
"states": [
   {
     "name": "start",
     "type": "callback",
     "action": {
        "name": "no operation",
        "functionRef": {
          "refName": "noop",
          "arguments": {}
        }
      },
      "eventRef": "cwm.forms.Check_applicant_age",
      "metadata": {
        "formData": "${ {user : .user} }",
        "taskName": "Provide applicant age"
      },
      "timeouts": {
        "eventTimeout": "PT15S",
        "stateExecTimeout": "PT15M",
        "actionExecTimeout": "PT15S"
      },
      "transition": "evaluate"
    },
    ...
 ]
```

Note that the callback state requires the `action` parameter to be stated. The `name` and `functionRef` keys need to have values provided even if no action is expected to happen during this state. Therefore, a mock action needs to be provided. For this, you can use a CWM utility function, like `noOp`, or any activity of your custom adapter, but keep in mind that the action needs to complete successfully for the workflow to pass to another step.

See the descriptions of utilities below to learn more about them.

FailJob

`system.function.@latest.common.FailJob`

The `FailJob` function allows a workflow creator to explicitly fail a workflow with a code and message when certain conditions are met, which currently is not provided for in the Serverless workflow specification.

GetResourceType

`system.function.@latest.resource.GetResourceType`

`GetResourceType` returns the Resource type for a given resource ID. You provide the resource ID using the `resourceId` parameter inside the `input` key under `arguments`. You can provide it explicitly or use a variable that you will pass as the input data for the workflow.

Example:

```
      "name": "begin",
      "type": "operation",
      "action": {
        "name": "getResType",
        "functionRef": {
          "refName": "GetResourceType",
          "arguments": {
            "input": {
                "resourceId": "${ .resID }"
            }
          }
        }
      },
```

NoOp

```
system.function.@latest.common.NoOp
```

`NoOp` is used to perform a mock action during workflow execution. A great example of a use case for `NoOp` is the `Callback` state, which mandates that an action be invoked before the callback state waits for an event. It is perfectly valid for a `Callback` state to execute a mock action and just wait for an event which is capturing data from the user. The `NoOp` function will help bypass the action and just wait for the defined event.

WaitUntil

```
system.function.@latest.common.WaitUntil
```

`WaitUntil` introduces a pause in workflow execution based on a user-provided timestamp. The timestamp is the time until which the workflow should wait provided in the RFC 3339 standard format using the `timestamp` parameter inside `arguments`.

Example:

```
"name": "begin",
"type": "operation",
"action": {
  "name": "WaitUntil",
  "functionRef": {
    "refName": "WaitUntil",
    "arguments": {
      "timestamp": "2024-09-19T16:32:37+02:00"
    }
  }
},
```

Expression functions

Expression functions are reusable logic blocks defined in a workflow. They evaluate specific conditions or expressions using workflow or state data. You can reference them in different workflow states by their name.

Defining an expression function

Inside a workflow definition, functions are defined with a `type: expression` parameter and an operation that specifies the logic in the form of a jq expression.

For example:

```
"functions": [
  {
    "name": "is-adult",
    "operation": ".applicant | .age >= 18",
    "type": "expression"
  },
]
```

Workflow states can call these functions to make decisions. For instance, a switch state can use `"is-adult"` to decide if the application should be approved or rejected based on age. Expression functions can also be used in state actions to perform calculations. For instance, an action increments a counter by 1 using the `"increment-count-function"`. The workflow starts with a count of 0, and after the function runs, the count becomes 1.

Example workflow

```
{
 "id": "fillglassofwater",
 "name": "Fill glass of water workflow",
 "version": "1.0",
 "specVersion": "0.8",
 "start": "Check if full",
 "functions": [
  {
   "name": "Increment Current Count Function",
   "type": "expression",
   "operation": "${.counts.current += 1 | .counts.current}"
  }
 ],
 "states": [
  {
   "name": "Check if full",
   "type": "switch",
   "dataConditions": [
    {
     "name": "Need to fill more",
     "condition": "${ .counts.current < .counts.max }",
     "transition": "Add Water"
    },
    {
     "name": "Glass full",
     "condition": "${ .counts.current >= .counts.max }",
     "end": true
    }
   ],
   "defaultCondition": {
    "end": true
   }
  },
  {
   "name": "Add Water",
   "type": "operation",
   "actions": [
    {
     "functionRef": "Increment Current Count Function",
     "actionDataFilter": {
      "toStateData": "${ .counts.current }"
     }
    }
   ],
   "transition": "Check if full"
  }
 ]
}
```

The expression function in this workflow increments the current count of water added to the glass and checks it against the maximum count, ensuring the process of filling is controlled and stops once the glass is full.

Input schema validation

The **Input schema** forms a contract between workflows and the entities that interact with them. If data that is provided to a workflow does not conform to the input schema, this could potentially result in catastrophic errors during execution. Validating the workflow upfront means the operator can be informed beforehand and get immediate feedback and ability to rectify any potential issues due to invalid data input.

The Serverless workflow specification 0.9 allows a workflow definition to specify the `dataInputSchema` parameter. The schema uses JSON Schema specification to define what inputs are required for the workflow to execute. CWM 1.2 supports the input schema validation for a workflow definition to ensure that when a new job is created, input data is validated against the `dataInputSchema`. If input data for a job is not valid, an error is returned.

Here is how you define an input schema inside a workflow definition.

Define input schema for workflow validation

To use an input schema to validate a workflow, simply include a valid JSON schema in a workflow using the `dataInputSchema` parameter. For example:

```
{
  "dataInputSchema": {
    "schema": {
      "title": "MyJSONSchema",
      "properties": {
        "firstName": {
          "type": "string"
        },
        "lastName": {
          "type": "string"
        }
      }
    },
    "failOnValidationErrors": true
  }
}
```

The `dataInputSchema` itself is validated when a workflow is added or modified. This applies to both creating and importing workflow definitions. On the other hand, the input data for a job is validated before the job is run. This applies to both immediate and scheduled job execution.

Tip You can turn off the validation while leaving the input schema in the workflow definition by setting the `failOnValidationErrors` parameter to `false`.

Required keys

In the schema specification, all the keys and values are optional by default. To make them mandatory, you need to specify them under the `required` key. For exampl3e:

```
"dataInputSchema": {
    "schema": {
      "title": "MyJSONSchema",
      "required": [
        "deviceName"
      ],
      "properties": {
        "deviceName": {
          "type": "string"
        },
        "nsoResource": {
          "type": "string"
        }
      }
    },
    "failOnValidationErrors": true
}
```

In this example, the `deviceName` key is now required. If the input data does not contain it, the job won't start and a validation error will be returned.

CHAPTER 5

Function definitions

This section contains the following topics:

• Function definitions, on page 37

Function definitions

Function definitions describe the functions available for the workflow to execute and the name of the adapter and activity that should be invoked by the engine when that function is invoked. While the Serverless Workflow specification supports various types of functions, CWM supports only those custom type functions that map to activities exposed via Adapters.

The following properties of function definitions are supported.

Table 11: Function definitions

Parameter	**Definition**
name	Name of function definition.
operation	Defines the adapter name and activity name that should be invoked by the engine. Format is <adapter name>.<activity name>. For example: The NSO Adapter has an activity called `RestconfGet`. An operation for this would be the name of the activity as registered in the worker, such as `RestconfGet`. Note that this name is case-sensitive.
metadata	Allows modelling of information beyond the core definition of the Serverless Workflow specification. The "worker" key is used to define which Taskqueue the activities will be executed on. CWM supports the concept of Workers that execute an Activity and are assigned Taskqueues that they listen to. To schedule an activity to run, the workflow engine places the activity on a Taskqueue. A worker process picks up the tasks to execute from the Taskqueue and executes the activity.

Example:

```
"functions": [
        {
            "name": "NSO.RestconfGet",
            "operation": "restconf_Get"
        "metadata":  {
            "worker": "defaultWorker"
        }
    },
    {
        "name": "NSO.RestconfPut",
        "operation": "restconf_Put"
        "metadata":  {
            "worker": "defaultWorker"
        }
    },
    {
        "name": "NSO.RestconfPost",
        "operation": "restconf_Post"
        "metadata":  {
            "worker": "defaultWorker"
        }
    },
    {
        "name": "NSO.RestconfPatch",
        "operation": "restconf_Patch"
        "metadata":  {
            "worker": "defaultWorker"
        }
    },
    {
        "name": "NSO.RestconfDelete",
        "operation": "restconf_Delete"
        "metadata":  {
            "worker": "defaultWorker"
        }
    },
    {
        "name": "NSO.SyncFrom",
        "operation": "device_SyncFrom"
        "metadata":  {
            "worker": "defaultWorker"
        }
    },
    {
        "name": "REST.Post",
        "operation": "rest_Post"
        "metadata":  {
            "worker": "defaultWorker"
        }
    } ]
```