

Using the VMware vRealize Orchestrator Client

04 FEBRUARY 2021
vRealize Orchestrator 8.3

You can find the most up-to-date technical documentation on the VMware website at:

<https://docs.vmware.com/>

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2008-2021 VMware, Inc. All rights reserved. [Copyright and trademark information.](#)

Contents

- 1 Using the VMware vRealize Orchestrator Client 6**
- 2 Introduction to the vRealize Orchestrator Client 7**
 - vRealize Orchestrator Client Usage Dashboard 8
 - Content Organization in the vRealize Orchestrator Client 8
 - Create Folders or Subfolders 9
 - Move Objects and Folders 10
 - Delete Folders or Subfolders 11
- 3 Setting Up the vRealize Orchestrator Client 12**
 - vRealize Orchestrator Roles and Groups 12
 - Assign Roles in the vRealize Orchestrator Client 14
 - Configure vRealize Orchestrator Client Roles in vRealize Automation 14
 - Create Groups in the vRealize Orchestrator Client 15
 - vRealize Orchestrator Object Version History 16
 - Restore a Workflow to an Earlier Version 16
 - Visual Comparison Between Workflow Versions 17
 - Reset Your vRealize Orchestrator Content Inventory to a Previous State with Git 18
- 4 vRealize Orchestrator Use Cases 19**
 - How to Integrate Amazon Web Services in vRealize Orchestrator by Using Python 19
 - Create Initial Python Script 20
 - Create the Amazon Web Services Action 21
 - Debug the Amazon Web Services Action 22
 - Update the Amazon Web Services Action 25
 - How Can I Use Git Branching to Manage My vRealize Orchestrator Object Inventory 26
 - Prepare Your GitLab Environment 27
 - Configure a Connection to a Git Repository 27
 - Push Changes to a Git Repository 29
 - How Can I Use Third-Party Modules to Call the vRealize Automation Project API 31
 - Create a Python Script That Calls the vRealize Automation Project API 31
 - Create a Node.js Script That Calls the vRealize Automation Project API 33
 - Create a PowerShell Script That Calls the vRealize Automation Project API 35
- 5 Managing Workflows 39**
 - Standard Workflows in the vRealize Orchestrator Workflow Library 40
 - Create Workflows 40
 - Edit Workflows and Actions from the Parent Workflow 40

- vRealize Orchestrator Input Form Designer 41
 - Create the Workflow Input Parameters Dialog Box in the vRealize Orchestrator Client 41
 - Input Parameter Properties in the vRealize Orchestrator Client 42
 - Using Actions to Validate vRealize Orchestrator Workflow Inputs 43
 - Requests for User Interaction in the vRealize Orchestrator Client 44
 - Schedule Workflows 45
 - Edit Scheduled Task in the vRealize Orchestrator Client 45
 - Find Object References in Workflows 46

- 6 Managing Actions 48**
 - Create Actions 48
 - Running and Debugging Actions 49
 - Run Actions 49
 - Debug Actions 50
 - Core Concepts for Python, Node.js, and PowerShell Scripts 51
 - Runtime Limits for Python, Node.js, and PowerShell Scripts 52

- 7 Managing Configuration Elements 54**
 - Create Configuration Elements 54

- 8 Managing Policies 56**
 - Create and Apply Policies 56
 - Policy Elements 57
 - Manage Policy Runs 58

- 9 Managing Resource Elements 59**

- 10 Managing Packages 60**
 - Create Packages 60
 - Export Packages 61
 - Import Packages 62

- 11 Troubleshooting in the vRealize Orchestrator Client 64**
 - Metric Data in the vRealize Orchestrator Client 64
 - Profile Workflows in the vRealize Orchestrator Client 64
 - Using the vRealize Orchestrator System Dashboard 65
 - Using Workflow Token Replay in the vRealize Orchestrator Client 66
 - Validating vRealize Orchestrator Workflows 67
 - Validate a Workflow and Fix Validation Errors in the vRealize Orchestrator Client 67
 - Debug Workflow Scripts in the vRealize Orchestrator Client 68
 - Debug Workflows by Schema Element 69

Configuring a Photon OS Container for Python Packages 70

Using the VMware vRealize Orchestrator Client

1

Using the VMware vRealize Orchestrator Client provides information about the workflow automation features and functionality of the vRealize Orchestrator Client.

Intended Audience

This information is intended for experienced system administrators who are looking for a tool that can help them to run and manage vRealize Orchestrator workflows.

Introduction to the vRealize Orchestrator Client

2

Use the vRealize Orchestrator Client to manage vRealize Orchestrator services and objects.

You can access the vRealize Orchestrator Client at `https://your_orchestrator_FQDN/orchestration-ui`.

UI element	Description
Dashboard	Use the vRealize Orchestrator Client Dashboard and profiling feature to gather useful metric data about your vRealize Orchestrator environment and workflows.
Workflows	Create, edit, schedule, run, and delete workflows.
Actions	Create, edit, and delete actions. The action editor supports automatic completion for common script elements included in the vRealize Orchestrator API Explorer.
Policies	Create, edit, run, and delete policies.
Packages	Create, delete, export, and import packages containing vRealize Orchestrator objects.
Configurations	Create, run, and delete configuration elements.
Resources	Export, import, and update resource elements.
Groups	Users with administrator rights can assign roles to users in the vRealize Orchestrator Client and add them to groups.
Git Repositories	Create an integration to a Git repository and use the integration to manage the development of workflows and other vRealize Orchestrator across multiple deployments. See How Can I Use Git Branching to Manage My vRealize Orchestrator Object Inventory .
API Explorer	Explore the API commands available in the vRealize Orchestrator Client. Note The vRealize Orchestrator Client communicates with the vRealize Orchestrator REST API through a REST proxy.

This chapter includes the following topics:

- [vRealize Orchestrator Client Usage Dashboard](#)
- [Content Organization in the vRealize Orchestrator Client](#)

vRealize Orchestrator Client Usage Dashboard

The vRealize Orchestrator Client dashboard provides a useful tool for monitoring, managing, and troubleshooting vRealize Orchestrator Client workflows.

Information on the vRealize Orchestrator Client dashboard is spread among five panels.

Window	Description
Workflow runs	Provides visual data about the number of running, waiting, and failed workflow runs.
Favorite workflows	Displays workflows added to favorites.
Waiting for input	Displays pending workflow runs that require further user interaction. These workflows are also displayed in the notifications menu in the upper-right corner of the UI.
Recent workflow runs	Manage recent workflow runs. Shows the name, state, start date, and end date of the workflow run.
Requiring Attentions	Displays failed workflow runs and workflow run performance metrics.

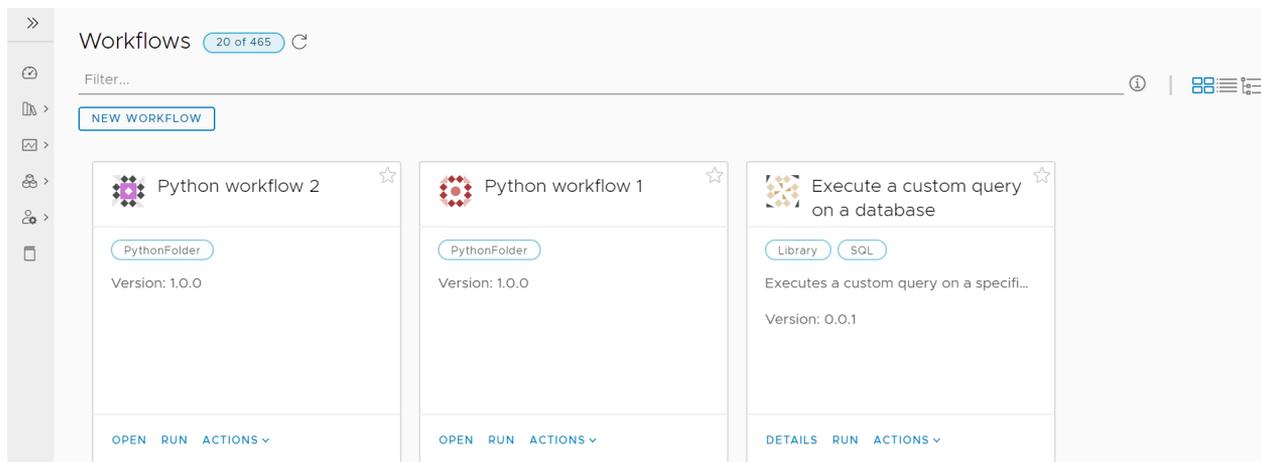
Content Organization in the vRealize Orchestrator Client

Manage how your vRealize Orchestrator object inventory is displayed in the vRealize Orchestrator Client.

The vRealize Orchestrator Client supports three different view types for objects such as workflows, actions, policies, resources, and configurations: Card View, List View, and Tree View. You can change the current view type from the top-right corner of the page.

Card View

The Card View is the default view type used in the vRealize Orchestrator Client. Information on the individual inventory object, such as a workflow, is displayed in a separate card element.



List View

List View displays information on your vRealize Orchestrator objects organized as a list. For more information on the actions you can perform on the object, click the vertical ellipses icon to the left of the object.

The screenshot shows the 'Workflows' list view in the vRealize Orchestrator Client. The page title is 'Workflows' with a count of '465 of 465'. There is a search bar labeled 'Filter...' and a 'NEW WORKFLOW' button. The main content is a table with the following data:

Name	Tags	Version	Description
Python workflow 2	PythonFolder	1.0.0	
Python workflow 1	PythonFolder	1.0.0	
Execute a custom query on a database	Library SQL	0.0.1	Executes a custom query on a specified database and returns the number of affected rows. You can run the workflow to update, delete, and insert queries.
JDBC URL generator	Library JDBC	0.0.8	Solicits information to generate a connection URL for JDBC database connections. The workflow emits the connection string it generates as output via the system log, and confirms the string can create a connection to the specified database.

Tree View

You can organize your object inventory under hierarchical folders in Tree View. Each vRealize Orchestrator object type has a root level folder. You cannot create new objects, such as workflows, under the root folder. You must create separate folders organized under the root folder. Each folder includes tools to help you manage its content, such as a content filter.

Note Each folder has a separate content filter. You cannot filter content across folders.

For more information on folders, see [Create a Folder or Subfolder in the vRealize Orchestrator Client](#).

Note When you select an object from the Tree View, it opens in a read-only mode. To edit the object content, such as workflow variables or the workflow schema, click **Edit** from the top options menu.

The screenshot shows the 'Tree View' in the vRealize Orchestrator Client. The left sidebar shows a hierarchical tree structure with 'Workflows' expanded to show 'Library' and 'PythonFolder'. 'PythonFolder' is selected and highlighted. The main content area shows the details for 'PythonFolder', including a description: 'A folder for Python-based workflows.' Below the description is a search bar labeled 'Filter...' and a toolbar with buttons for '+ NEW FOLDER', '+ NEW WORKFLOW', '% MOVE', and 'X DELETE'. A table lists the contents of the folder:

Name	Type
Python workflow 1	Workflow
Python workflow 2	Workflow

At the bottom right, there is a pagination control showing 'Items per page 20' and '1 - 2 of 2 items'.

Create a Folder or Subfolder in the vRealize Orchestrator Client

Organize your vRealize Orchestrator objects by using a hierarchical folder structure.

You can create folders and subfolders to organize the following types of vRealize Orchestrator objects:

- Workflows
- Actions
- Policies
- Configuration elements
- Resource elements

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 From the left navigation pane, select an object page, such as **Workflows**.
- 3 From the top-right, select the tree view icon () .
- 4 (Optional) To create a subfolder, select a parent folder from the tree view on the left.
- 5 Click **New Folder**.
- 6 Enter a name and description, and click **Save**.
- 7 Add objects or subfolders to the newly created folder.
- 8 (Optional) To edit the folder name, select **Actions > Edit**.

Move Objects and Folders in the vRealize Orchestrator Client

Reorganize your vRealize Orchestrator content by moving the content into another folder.

You cannot move actions between action modules, or move any objects to a root folder. The root folder includes the main object folders and subfolders, but cannot be used to store objects.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 From the left navigation pane, select an object page, such as **Workflows**.
- 3 From the top-right, select the tree view icon () .
- 4 Expand the tree view, and select the object or folder you want to move.
- 5 Drag the object or folder to its new parent folder.

Note You can also move objects into new folders directly from the object editor. On the **Summary** tab, click **Select Folder**, and select the new parent folder for the object. Another move option is to select objects from the table on the folder page. This option is useful for performing batch move operations that include multiple vRealize Orchestrator objects.

Delete a Folder or Subfolder in the vRealize Orchestrator Client

Delete obsolete folders or subfolders from your vRealize Orchestrator Client.

You cannot delete the corresponding root-level folder of each vRealize Orchestrator object type.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 From the left navigation pane, select an object page, such as **Workflows**.
- 3 From the top-right, select the tree view icon () .
- 4 Tick the check box next to the folder you want to delete.

Note To delete a subfolder, select the parent folder from the tree view and then tick the check box.

- 5 Click **Delete**.
- 6 If the selected folder is empty.
 - a Confirm that you want to delete to folder.
 - b Click **Delete**.
- 7 If the selected folder contains vRealize Orchestrator Client objects or subfolders.
 - a Confirm that you want to delete the folder.
 - b Click **Delete**.

You receive the message `Could not delete item 'your_folder_name': Folder 'your_folder_name' is not empty.`
 - c To delete the folder and all its content, click **Force delete**.
 - d Confirm that you want to delete the folder, and click **Delete**.

Note You can also perform a batch delete by selecting multiple objects from the table included in the folder menu.

Setting Up the vRealize Orchestrator Client

3

To take full advantage of the functionality of the vRealize Orchestrator Client, you must configure your user permissions and learn how you can use version history to manage your objects.

This chapter includes the following topics:

- [vRealize Orchestrator Roles and Groups](#)
- [vRealize Orchestrator Object Version History](#)

vRealize Orchestrator Roles and Groups

vRealize Orchestrator administrators can set permissions that control access to features and content in the vRealize Orchestrator Client. Access rights are separated into user roles and group permissions.

Roles control what vRealize Orchestrator Client features users can view and use. Access to the role management functionality depends on the license type of your vRealize Orchestrator environment.

Table 3-1. License-Based Access to vRealize Orchestrator Role Management

License	Authentication	
	vSphere	vRealize Automation
vSphere	Role management is not supported. Groups support only Run permissions.	
vRealize Automation	Manage roles in the vRealize Orchestrator Client. See Assign Roles in the vRealize Orchestrator Client .	Manage roles through Identity and Access Management in vRealize Automation. See Configure vRealize Orchestrator Client Roles in vRealize Automation .

Group permissions control what vRealize Orchestrator Client content users can view and use, such as workflows, actions, policies, configuration elements, and resource elements. Access to preconfigured system vRealize Orchestrator content like standard workflows and actions is shared among all users, unless configured otherwise through group permissions.

Access rights of users with administrator and viewer roles are not restricted by group permissions. Access rights of users without an assigned role and users with a workflow designer role depend on the group assigned to them. You can extend the access rights of these users by modifying their group permissions. In this way, you can organize users into common projects. For example, you can create a group that includes users working on developing a custom vRealize Orchestrator plug-in and allow them to modify only content that is specific to their group.

Table 3-2. vRealize Orchestrator User Roles and Groups Permissions

Role	Access Rights		
Administrator	Administrators can access all vRealize Orchestrator Client features and content, including the content created by specific groups. Responsible for setting user roles, creating and deleting groups, and adding users to groups. Administrators are not limited by group permissions.		
	<p>Note Tenant administrators from vRealize Automation environments used to authenticate vRealize Orchestrator have Administrator rights by default.</p>		
Viewer	Viewers have read-only access to all content in the vRealize Orchestrator Client, but cannot create, edit, run, or export content. Viewers can also see all groups and group content. Viewers are not limited by group permissions.		
	Group Permissions		
	No assigned group	Run	Run and edit
Workflow Designer	<ul style="list-style-type: none"> ■ View system content. ■ View and run own runs. ■ Create, run, edit, and delete own content. 	<ul style="list-style-type: none"> ■ View system content ■ View and run own runs. ■ Create, run, edit, and delete own content. ■ Add own content to the group. ■ Run group content, but cannot edit it. 	<ul style="list-style-type: none"> ■ View system content. ■ View and run own runs. ■ Create, run, edit, and delete own content. ■ Add own content to the group. ■ Run and edit group content. <p>Note Not available for vRealize Orchestrator instances authenticated with vSphere.</p>
User without an assigned role	<ul style="list-style-type: none"> ■ View own runs. 	<ul style="list-style-type: none"> ■ View and run own runs. ■ View and run group content. 	<ul style="list-style-type: none"> ■ View and run own runs. ■ View and run group content. <p>Note To be able to create, edit, and add content, users in this group must be assigned a Workflow Designer role.</p> <p>Note Not available for vRealize Orchestrator instances authenticated with vSphere.</p>

Assign Roles in the vRealize Orchestrator Client

As an administrator, you can add users to the vRealize Orchestrator Client and set what features they can view and use.

Role management controls the access of users from the vRealize Orchestrator identity provider to the features of the vRealize Orchestrator Client. Role management covers both the vRealize Orchestrator Client user interface and the API functionality.

Note Client-side role management is only available for vRealize Orchestrator instances authenticated with vSphere that use a vRealize Automation license. For information on assigning roles to vRealize Orchestrator authenticated with vRealize Automation, see [Configure vRealize Orchestrator Client Roles in vRealize Automation](#).

Procedure

- 1 Log in to the vRealize Orchestrator Client as an administrator.
- 2 Navigate to **Administration > Roles Management**.
- 3 Click **Add**.
- 4 Search for the user or group you want to add to the vRealize Orchestrator Client.
- 5 Select the user's role. For more information on roles, see [vRealize Orchestrator Roles and Groups](#).
- 6 Click **Save**.

Configure vRealize Orchestrator Client Roles in vRealize Automation

You can assign service roles for the vRealize Orchestrator Client in the **Identity & Access Management** page in vRealize Automation. Service roles can be assigned for the embedded vRealize Orchestrator Client and standalone vRealize Orchestrator instances authenticated with vRealize Automation.

vRealize Orchestrator service roles manage what features of the embedded vRealize Orchestrator Client users can access. For more information vRealize Orchestrator roles, see [vRealize Orchestrator Roles and Groups](#).

Note Standalone vRealize Orchestrator instances authenticated with vSphere that use a vRealize Automation license can assign roles directly in the vRealize Orchestrator Client. See [Assign Roles in the vRealize Orchestrator Client](#).

Prerequisites

- Verify that appropriate users and groups are imported from a valid vIDM instance.
- Before assigning a vRealize Orchestrator service role to your user, verify that your user has an assigned organization role in vRealize Automation. See, *Administering Users and Groups in vRealize Automation* in *Administering vRealize Automation*.

Procedure

- 1 From the top-right header drop-down menu, select the **Identity & Access Management** option.
- 2 On the **Active Users** tab, search for the email address of the user you want to assign to vRealize Orchestrator.
- 3 Select the check box next to the user, and click **Edit Roles**.
- 4 Click **Add Service Access**.
- 5 From the left drop-down menu, select **Orchestrator**.
- 6 From the right drop-down menu, select the role you want to assign to the user.
- 7 Click **Save**.

Create Groups in the vRealize Orchestrator Client

As an administrator, you can use groups to set what vRealize Orchestrator content users can view and access in the vRealize Orchestrator Client.

You can use the vRealize Orchestrator Client to set group permissions to vRealize Orchestrator workflows, actions, policies, configuration elements, resource elements, and packages.

Note Users from vRealize Orchestrator instances authenticated with vSphere, can only have **Run** group permissions.

Procedure

- 1 Log in to the vRealize Orchestrator Client as an administrator.
- 2 Navigate to **Administration > Groups**.
- 3 Click **New Group**.
- 4 On the **Summary** tab, add a name and description for the group.
- 5 On the **Users** tab, click **Add**.
 - a Search for a user you want to add to the group.
 - b Assign group permissions to the user.
 - c Click **Add**.
- 6 On the **Items** tab, add vRealize Orchestrator objects to the group.

Note You can also add an object to existing groups when that object is being created in the vRealize Orchestrator Client. To add the object, select the group from the **Accessible by** drop-down menu on the **Summary/General** tab of the object editor.

- 7 Click **Save**.

vRealize Orchestrator Object Version History

The vRealize Orchestrator Client maintains a version history record for each vRealize Orchestrator object. Using the version history, you can compare different vRealize Orchestrator object versions and revert to a previous version.

vRealize Orchestrator creates a version history record for each vRealize Orchestrator object when you save the object. Subsequent changes to vRealize Orchestrator objects create a new version history record. The previous versions history records are preserved and can be used to track changes to the object and revert the object to a previous version. Reverting an object to a previous version creates a new version history record.

The vRealize Orchestrator Client tracks the version history of the following vRealize Orchestrator objects:

- Workflows
- Actions
- Packages
- Policies
- Configuration elements

Note Generated workflows do not appear in the workflow version history. For example, the workflows generated by the **Generate CRUD workflows for a table** workflow do not appear on the **Version History** tab and cannot be pushed to any configured Git repositories. To include these workflows in the vRealize Orchestrator version history, duplicate the generated workflows.

You can access the version history of an object from the **Version History** tab of the object editor page. If you are attempting to edit an object at the same time as another user, a merge conflict might occur. To resolve the merge conflict, click **Resolve** to the right of the error message. On the **Resolve Conflicts** window you have three options:

- **Use theirs.** Resolve the merge conflict by using the changes made by the other user.
- **Use ours.** Resolve the merge conflict by using your changes.
- **Resolve.** Resolve the merge conflict by editing the displayed change model. If the provided model is invalid, this option is unavailable.

Restore a Workflow to an Earlier Version

You can restore a workflow to a previously saved version.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Workflows**, and select a workflow.
- 3 Select the **Version History** tab.

- 4 To view a comparison between the versions, select a workflow version and select another version from the **Diff against** drop-down menu.

A window displays the differences between the current workflow version and the selected workflow version.

- 5 To restore the workflow to another version, click **Restore**.

The workflow state is reverted to the state of the selected version.

Note You can also restore a workflow version from the graphic difference tool view. See [Visual Comparison Between Workflow Versions](#).

Visual Comparison Between Workflow Versions

Compare changes between workflow versions with the graphic difference tool.

By default, the vRealize Orchestrator Version History displays differences between workflow versions in a YAML form. You can also perform a visual comparison between different workflow versions. You can view changes in:

- The general workflow information, such as version number and workflow description.
- The variables used in the workflow.
- The input and output parameters of the workflow.
- The workflow schema.

Prerequisites

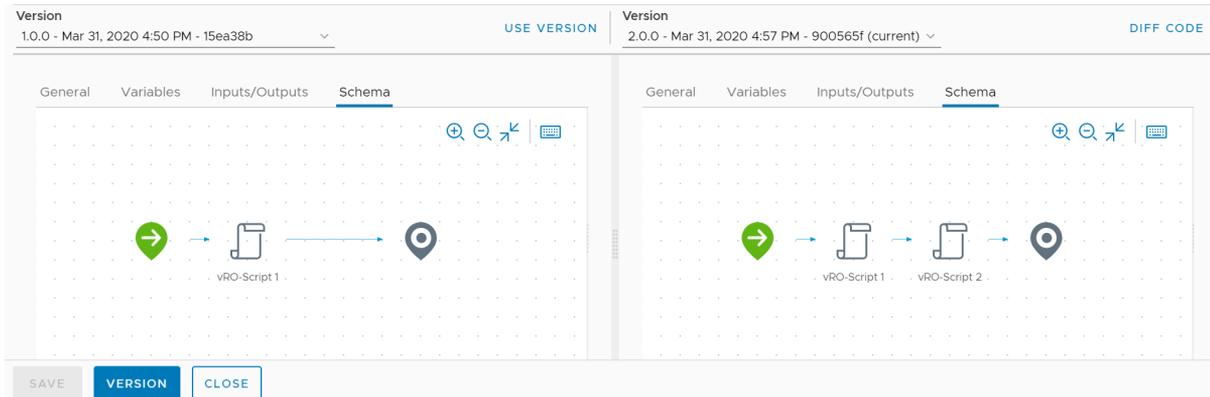
Create a workflow.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Workflows**, and select one of your workflows.
- 3 Edit the content of the workflow.
For example, you can add an extra **Scriptable task** element on the **Schema** tab.
- 4 Click **Save**.
- 5 Select the **Version History** tab.

6 From the top-right, select **Diff Visually**.

You can now perform a visual comparison between two selected workflow versions. You can select which versions to compare from the **Version** drop-down menu.



7 (Optional) You can restore a workflow to another version by selecting **Use Version**.

Reset Your vRealize Orchestrator Content Inventory to a Previous State with Git

By using an earlier Git commit, you can reset your vRealize Orchestrator content to an earlier state.

You can reset your vRealize Orchestrator content to a previous state by selecting a specific commit.

Prerequisites

- Configure a connection to a GitHub or GitLab repository. See [Configure a Connection to a Git Repository](#).
- Push a local change set to the configured Git repository.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Administration > Git History**.
- 3 Select a change set you want to reset to and click **Reset to this**.
- 4 Confirm that you want to reset to this specific commit and click **Ok**.

The vRealize Orchestrator content inventory is reset to the state specified in the commit. Relevant vRealize Orchestrator content is reverted to a previous version. If the content did not exist when the commit was pushed, it is removed from the inventory.

What to do next

To restore the vRealize Orchestrator inventory to the latest state saved to the Git repository, perform a `pull` command from the **Git History** window.

vRealize Orchestrator Use Cases

4

These use cases demonstrate part of the functionality of the vRealize Orchestrator platform.

These use cases present example values only. Your own environment structure and naming conventions can vary.

This chapter includes the following topics:

- [How to Integrate Amazon Web Services in vRealize Orchestrator by Using Python](#)
- [How Can I Use Git Branching to Manage My vRealize Orchestrator Object Inventory](#)
- [How Can I Use Third-Party Modules to Call the vRealize Automation Project API](#)

How to Integrate Amazon Web Services in vRealize Orchestrator by Using Python

This vRealize Orchestrator use case shows an example of how you can use Python to expand the capabilities of your vRealize Orchestrator deployment.

Starting with vRealize Orchestrator 8.1, you can use three new runtimes for use in your actions and workflow scripts.

- Python 3.7
- Node.js 12
- PowerCLI 11/Powershell 6.2
- PowerCLI 12/Powershell 7

Note The PowerCLI runtime includes PowerShell and the following modules: VMware.PowerCLI, PowerNSX, PowervRA.

Important You can only use the new runtimes if your vRealize Orchestrator deployment uses a vRealize Automation license.

This use case demonstrates how you can create a Python script that calls up EC2 instances in Amazon Web Services (AWS).

Important Before you begin developing your custom script, verify that you are familiar with the core concepts of using Python, Node.js, and PowerShell scripts, in vRealize Orchestrator. See [Core Concepts for Python, Node.js, and PowerShell Scripts](#) .

Procedure

1 Create Initial Python Script

On your local machine, create your Python script and package the script and a boto3 library as a ZIP folder.

2 Create the Amazon Web Services Action

Create a vRealize Orchestrator action that uses that uses your Python script.

3 Debug the Amazon Web Services Action

The original version of the Python script has a deliberate built-in error, so you can learn how to debug your script.

4 Update the Amazon Web Services Action

Import the updated Python script, and run the action again.

Create Initial Python Script

On your local machine, create your Python script and package the script and a boto3 library as a ZIP folder.

Prerequisites

- Download and install Python 3. See the [Python Downloads](#) page.
- Download and Install Visual Studio Code. See the [Visual Studio Code download](#) page.
- Verify that you have installed the Python extension for Visual Studio Code. See the [Visual Studio Marketplace](#).

Procedure

- 1 On your local machine, create a `vro-python-aws` folder, and install the boto3 Python SDK on it.

```
mkdir vro-python-aws
cd vro-python-aws
mkdir lib
pip install boto3 -t lib/
```

- 2 Open an editor, and create the main Python script. For this use case, you are using Visual Studio Code.

```
import boto3

def handler(context, inputs):
    ec2 = boto3.resource('ec2')
    filters = [{
        'Name': 'instance-state-name',
        'Values': ['running']
    }]

    instances = ec2.instances.filter(Filters=filters)
    for instance in instances:
        print('Instance: ' + instance.id)
```

This Python script lists all running EC2 instances in a given region.

- 3 Save the created script as a `main.py` file in the `vro-python-aws` folder.
- 4 Log in to a command-line interface.
- 5 Navigate to the `vro-python-aws` folder.

```
cd vro-python-aws
```

- 6 Create a ZIP package that contains the Python script.

```
zip -r --exclude=*.zip -X vro-python-aws.zip .
```

Note You can also create the ZIP package by using a ZIP utility tool, such as 7-Zip.

Results

You have created the base Python script, and prepared it for import into your vRealize Orchestrator deployment.

Create the Amazon Web Services Action

Create a vRealize Orchestrator action that uses that uses your Python script.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Actions**.
- 3 Click **New Action**.
- 4 On the **General** tab, enter a name, module, and version number for the action.
- 5 On the **Script** tab, select **Python 3.7** as the runtime, and **Zip** as the script type.
- 6 Click **Import**.

- 7 Browse to the `vro-python-aws` folder, and select the ZIP package containing the Python script.
- 8 In the **Entry handler** text box, enter `main.handler`.

Note The entry handler of the action is based on the main script in the imported ZIP package. Because the main script is located in a file called `main.py`, and a function called `handler`, the entry handler must be `main.handler`. If you have titled your main script file differently, change the entry handler value accordingly.

- 9 Save the action, and click **Run**.

The action run encounters an error.

- 10 Select the **Logs** tab.

The logs of the action run display a `"botocore.exceptions.NoRegionError: You must specify a region."` error message. This is expected behavior, as the initial Python script does not define a region.

What to do next

Debug the Python script. See [Debug the Amazon Web Services Action](#).

Debug the Amazon Web Services Action

The original version of the Python script has a deliberate built-in error, so you can learn how to debug your script.

Prerequisites

Sign in to your Amazon Web Services (AWS) account, and create a IAM user specifically for this use case scenario. See [Creating an IAM User in Your AWS Account](#). The IAM user must have the following permissions:

```
"Effect": "Allow",
"Action": "ec2:DescribeInstances",
"Resource": "*"

```

Procedure

1 Prepare the vRealize Orchestrator Appliance.

Caution Do not debug scripts in your production vRealize Orchestrator deployment. Debug from a single node vRealize Orchestrator deployment, that you use for development and testing.

- a Log in to the vRealize Orchestrator Appliance command line over SSH as **root**.
- b Run the `vracli dev tools` command.
- c You are prompted to confirm that you want to continue. Enter **yes** to continue, or **no** to cancel.

Important By running the `vracli dev tools` command, you open the ports required to debug the Python script. You must leave the current SSH session open during the debug process.

2 Start the debug configuration.

- a Log in to the vRealize Orchestrator Client.
- b Open the AWS action, and click **Debug**.
The debug process begins, and the action run is suspended.
- c Select the **Debug Configuration** tab.
The tab contains a `.json` configuration that you can remotely attach to your IDE to debug the Python script.
- d Copy the configuration content manually, or click **Copy To Clipboard**.

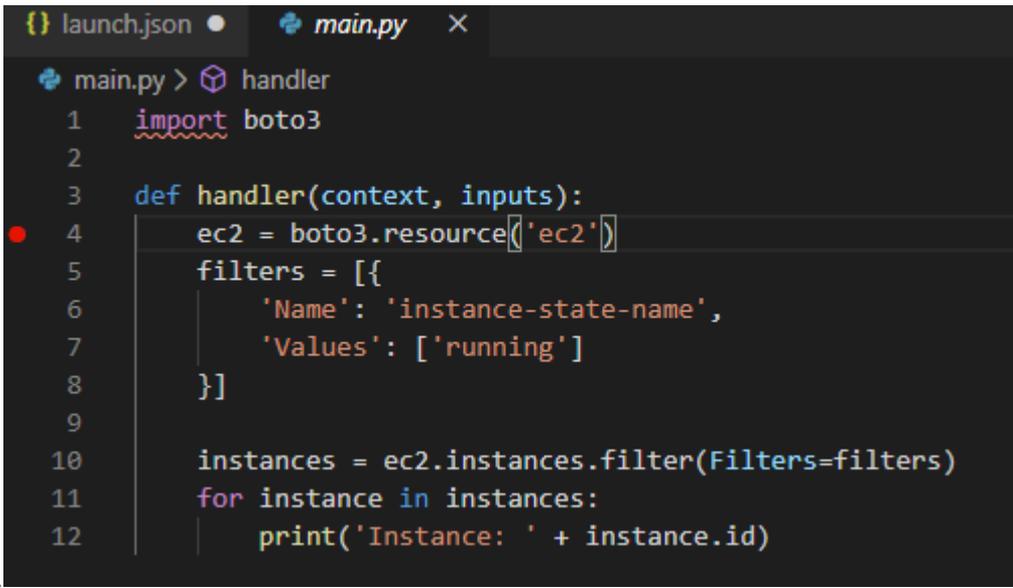
3 Debug the Python script.

- a Open Visual Studio Code.
- b Open the `vro-python-aws` folder.
- c From the top navigation pane, select **Run > Open Configurations**.
- d Select **Python file**.

- e Leave the "version" and "configuration" attributes in their current positions, and paste the content of the .json configuration copied from the vRealize Orchestrator Client. The generated launch.json file must look similar to this:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "request": "attach",
      "port": 18281,
      "name": "vRO Python debug 8302f4c7-5beb-40da-848a-5003c0296f7b",
      "host": "es-sof-vc-vm-225-190.sof-mbu.eng.vmware.com",
      "type": "python",
      "pathMappings": [
        {
          "localRoot": "${workspaceFolder}",
          "remoteRoot": "/var/run/vco-polyglot/function"
        }
      ]
    }
  ]
}
```

- f Select the main.py script file, and add a breakpoint to the `ec2 = boto3.resource('ec2')`



The screenshot shows a code editor with two tabs: 'launch.json' and 'main.py'. The 'main.py' tab is active, showing a Python script with a breakpoint (red dot) on line 4. The code is as follows:

```
main.py > handler
1 import boto3
2
3 def handler(context, inputs):
4     ec2 = boto3.resource('ec2')
5     filters = [{
6         'Name': 'instance-state-name',
7         'Values': ['running']
8     }]
9
10    instances = ec2.instances.filter(Filters=filters)
11    for instance in instances:
12        print('Instance: ' + instance.id)
```

line.

- g From the top navigation pane, select **Run > Start Debugging**.
- h When the debugger reaches the breakpoint, perform a step over operation.
The debug run indicates that the Python script lacks a specified region, and AWS access key.
- i Go back to the open vRealize Orchestrator Appliance session, and press **Enter** to close the ports you opened for this debug session.

4 Add the missing information to the Python script.

- a In Visual Studio Code, create a file called `awsconfig` that contains the AWS access key of the IAM user and the AWS region you want to ping with the Python script.

```
[default]
aws_access_key_id=your key ID
aws_secret_access_key=your secret access key
region=your-region
```

- b Save `awsconfig` as a configuration (`.cfg`) file in the `vro-python-aws` folder.
- c Open the `main.py` file, and modify it so the `boto3` library can use the `awsconfig.cfg` file.

```
import boto3

import os
os.environ['AWS_CONFIG_FILE'] = os.getcwd() + '/awsconfig.cfg'

def handler(context, inputs):
    ec2 = boto3.resource('ec2')
    filters = [{
        'Name': 'instance-state-name',
        'Values': ['running']
    }]

    instances = ec2.instances.filter(Filters=filters)
    for instance in instances:
        print('Instance: ' + instance.id)
```

- d Create a new ZIP package that contains the `main.py` file, `awsconfig.cfg` file, and `boto3` library.

```
zip -r --exclude=*.zip -X vro-python-aws.zip .
```

Note You can also create the ZIP package by using a ZIP utility tool, such as 7-Zip.

Update the Amazon Web Services Action

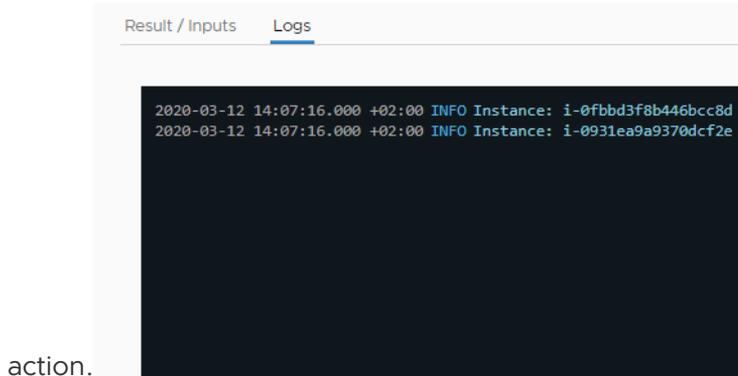
Import the updated Python script, and run the action again.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Actions**, and select the original Amazon Web Services (AWS) action.
- 3 (Optional) On the **General** tab, change the version number.
- 4 Remove the old ZIP package, and click **Import**.
- 5 Select the updated ZIP package.

- 6 Save the action, and click **Run**.
- 7 After the action run finishes, select the **Logs** tab.

The logs display the EC2 instances queried by the



action.

What to do next

Create a vRealize Orchestrator workflow that uses the updated AWS action as an **Action element**.

How Can I Use Git Branching to Manage My vRealize Orchestrator Object Inventory

Use branching to organize how you vRealize Orchestrator content is managed in your Git repository.

By using Git, you can increase the flexibility for your vRealize Orchestrator developers by providing a centralized repository. For example, you can use Git to manage the workflow development across multiple vRealize Orchestrator environments.

Note To use Git to manage your object inventory, your vRealize Orchestrator deployment must use a vRealize Automation license. For more information, see *vRealize Orchestrator Feature Enablement with Licenses in Installing and Configuring vRealize Orchestrator*.

Starting with vRealize Orchestrator 8.1, you can now push and pull objects to and from branches. You can use branching to manage the development of specific groups of vRealize Orchestrator objects, before they are merged back into your main branch.

In this use case, you are using a GitLab project to manage vRealize Orchestrator objects that use the Python runtime. This use case represents an example of the Git functionality in vRealize Orchestrator and does not represent the limits of the feature scope.

Note If you are more familiar with GitHub, you can use a GitHub repository for this use case.

Procedure

- 1 **Prepare Your GitLab Environment**

Create a Git branch for your vRealize Orchestrator Python objects.

2 Configure a Connection to a Git Repository

As an **administrator**, you can configure a connection between your vRealize Orchestrator deployment and a Git repository or project.

3 Push Changes to a Git Repository

Push your changes to local vRealize Orchestrator objects to your integrated Git repository. For this use case, we are pushing changes to a Python-based vRealize Orchestrator action to a specific Git branch.

Prepare Your GitLab Environment

Create a Git branch for your vRealize Orchestrator Python objects.

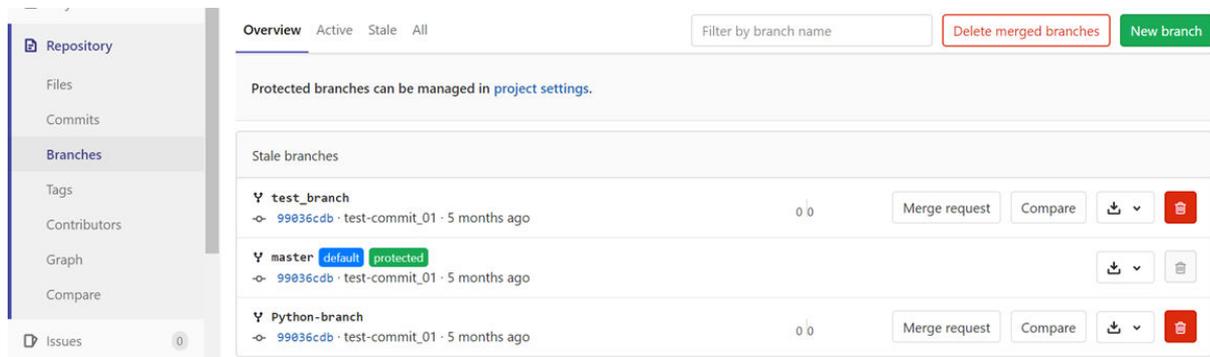
Prerequisites

Create a GitLab project for your vRealize Orchestrator environment. See [Create a project](#).

Procedure

- 1 Log in to your GitLab account.
- 2 Navigate to your GitLab project.
- 3 On the left navigation pane, select **Repository > Branches**.
- 4 On the **Overview** tab, click **New Branch**.
- 5 Under **Branch name**, enter **Python-branch**.
- 6 Leave the **Create from** option as **master**.
- 7 Click **Create branch**.

You have created a branch for your Python-based vRealize Orchestrator objects.



Configure a Connection to a Git Repository

As an **administrator**, you can configure a connection between your vRealize Orchestrator deployment and a Git repository or project.

To use Git for management of your vRealize Orchestrator object inventory, you must configure a connection to your Git repository by using the vRealize Orchestrator Client.

Prerequisites

- Verify that your vRealize Orchestrator environment uses a vRealize Automation license.
- Generate an access token for your GitLab project and copy it to your clipboard for use during the configuration process. See [Creating a personal access token](#).

Note For this use case, you are using a GitLab project. If you are more familiar with GitHub, you can use a GitHub repository. For information generating a GitHub token, see [Creating a personal access token for the command line](#).

Procedure

1 Log in to the vRealize Orchestrator Client as an **administrator**.

2 Navigate to **Administration > Git Repositories**.

3 Click **Add Repository**.

4 Enter the URL address of your Git repository.

For example, `https://gitlab.com/myusername/my-vro-repo`.

Note You can also establish a connection with the SSH protocol.

5 Enter the user name for your Git profile.

6 Enter the access token of your Git repository.

7 To validate the connection to the Git repository, click **Validate**.

8 (Optional) Change the name used to identify the repository in the vRealize Orchestrator Client.

9 (Optional) Add a short description for the connected Git repository.

10 To activate the connected Git repository, click **Make active repository**.

Note Only one Git repository can be active at a time. You can change the active Git repository from the **Git Repositories** page.

11 Select the branch to which you want to push your changes. For this use case, you are using **Python-branch**. See [Prepare Your GitLab Environment](#).

Note You can change the selected Git branch at any time after you finish the initial Git configuration.

12 To finish the configuration process, click **Save**.

What to do next

Navigate back to the **Git Repositories** menu and confirm that the status of the repository is **Active**.

Push Changes to a Git Repository

Push your changes to local vRealize Orchestrator objects to your integrated Git repository. For this use case, we are pushing changes to a Python-based vRealize Orchestrator action to a specific Git branch.

You can push a local change set to a Git repository. Each change set can consist of one or more modified vRealize Orchestrator objects.

Note The process of pushing and discarding change sets to a Git repository is not limited by group permissions. Therefore, a workflow developer from one group can push or discard local changes made by another developer.

Prerequisites

- Verify that you have created a Git branch. See [Prepare Your GitLab Environment](#).
- Verify that you have configured a connection with a Git repository. See [Configure a Connection to a Git Repository](#).
- Verify that your Git integration is set to push changes to the **Python-branch** Git branch.
- Create a Python-based vRealize Orchestrator object. For example, see [How to Integrate Amazon Web Services in vRealize Orchestrator by Using Python](#).

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Edit your Python action.
 - a Navigate to **Library > Actions**, and select your Python action.
 - b Make some minor changes to the action, such as changing the description.
 - c Save the action.

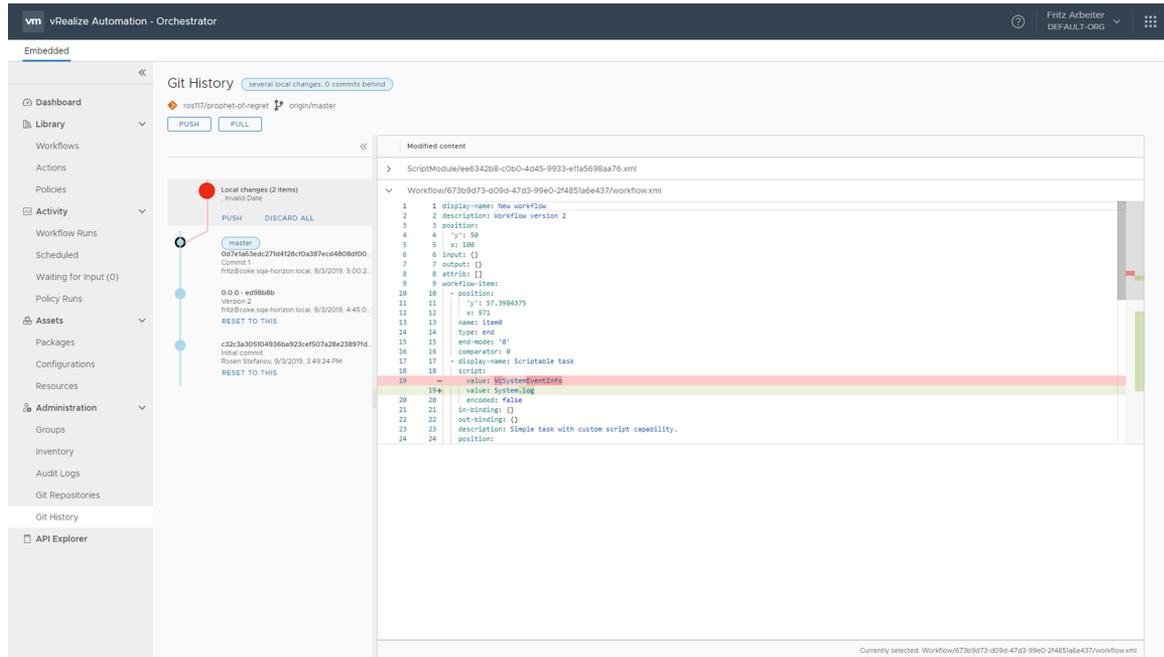
3 Push your changes to the Git repository.

Note You can also push local changes on a per object level by clicking the **Version** option displayed at the bottom of the object editor.

a Navigate to **Administration > Git History**.

Git History displays the current differences between the local version branch and the selected Git repository branch. You can expand the entry for any modified vRealize Orchestrator object to view the version differences.

Note You can discard a local change set by select **Discard all**.



b Click **Push**.

c Enter a commit title.

d (Optional) Enter a short description for the commit.

e Select the changes to your Python action that you want to push to the Git repository.

4 To finish pushing the local change set to the Git repository, click **Push**.

What to do next

After you finish development in your Git branch, merge it with the main branch. See [How to create a merge request](#).

How Can I Use Third-Party Modules to Call the vRealize Automation Project API

This vRealize Orchestrator use case demonstrates how you can call up vRealize Automation Project API by using third-party modules.

Starting with vRealize Orchestrator 8.1, you can use the following runtimes:

- Python 3.7
- Node.js 12
- PowerCLI 11/Powershell 6.2
- PowerCLI 12/Powershell 7

Note The PowerCLI runtime includes PowerShell and the following modules: VMware.PowerCLI, PowerNSX, PowervRA.

In this use case, you learn how to create vRealize Orchestrator actions that use third-party dependency modules to connect to the vRealize Automation Project API.

Important Before you begin developing your custom script, verify that you are familiar with the core concepts of using Python, Node.js, and PowerShell scripts, in vRealize Orchestrator. See [Core Concepts for Python, Node.js, and PowerShell Scripts](#) .

Create a Python Script That Calls the vRealize Automation Project API

Create a sample script that uses Python to call the vRealize Automation Project API.

Prerequisites

Verify that you have installed Python 3 and the PIP package installer. See [Python Downloads page](#) and [Python Package Index](#).

Procedure

- 1 On your local machine, open a command-line shell.
- 2 Create a `vro-python-vra` folder.

```
mkdir vro-python-vra
```

- 3 Navigate to the `vro-python-vra` folder.

```
cd vro-python-vra
```

- 4 Create a Python script called `handler.py`.

```
touch handler.py
```

The `handler.py` script must define one function that accepts two arguments, the context of the vRealize Orchestrator workflow run, and the bound vRealize Orchestrator inputs.

```
def handler(context, inputs):
    print('Hello, your inputs were ' + inputs)
    return None
```

Note Using standard logging libraries, everything you log in the action that uses your script is also shown in the workflow log. The inputs and the return of your script must have corresponding input parameters and return types configured in the vRealize Orchestrator Client. For example, the `vRAUrl` input, in your script must have a corresponding input parameter called `vRAUrl` in the vRealize Orchestrator Client. Similarly, if your script returns a string value, the return type configured in the vRealize Orchestrator Client must also be a string type. If your action returns a complex object, you can use `Properties` or `Composite Type` return type.

5 Install the Python requests module.

Important Third-party dependency modules must be installed in a root level folder in your main `vro-python-vra` script folder. For this use case, you create a `lib` folder for your requests module.

a Create a `lib` folder.

```
mkdir lib
```

b Install the requests module.

```
pip3 install requests -t lib/
```

6 Add the requests module to the `handler.py` script.

```
import requests

def handler(context, inputs):
    print('Hello, your inputs were ' + inputs)
    return None
```

7 Create a GET request to the vRealize Automation Project API.

```
token = ''
vRAUrl = ''
r = requests.get(vRAUrl + '/iaas/api/projects', headers={'Authorization': 'Bearer ' +
token})

print('Got response ' + r.text)
```

8 Define the `token` and `vRAUrl` values.

- a Retrieve the access token by using the vRealize Automation Identity Service API. See [Get Your Access Token for the vRealize Automation API](#)
- b For the `vRAUrl` value, define the script so it uses a vRealize Orchestrator input parameter with the same name.

```
vRAUrl = inputs["vRAUrl"]
```

- c Add the new values to the `handler.py` file.

```
import requests

def handler(context, inputs):
    token = 'ACCESS_TOKEN'
    vRAUrl = inputs["vRAUrl"]

    r = requests.get(vRAUrl + '/iaas/api/projects', headers={'Authorization': 'Bearer ' + token})

    print('Got response ' + r.text)

    return r.json()
```

Note Because the response from the vRealize Automation Project API is returned in a JSON format, use a `Properties or Composite Type` return type for your vRealize Orchestrator action.

- 9 Create a ZIP package that contains the `handler.py` file and `lib` folder of your request module.

```
zip -r --exclude=*.zip -X vro-python-vra.zip .
```

What to do next

Import the PowerShell script into a vRealize Orchestrator action. See [Create Actions in the vRealize Orchestrator Client](#).

Create a Node.js Script That Calls the vRealize Automation Project API

Create a sample script that uses Node.js to call the vRealize Automation Project API.

Prerequisites

Download and install Node.js 12. See [Node.js Downloads](#).

Procedure

- 1 On your local machine, open a command-line shell.

2 Create a `vro-node-vra` folder.

```
mkdir vro-node-vra
```

3 Navigate to the `vro-node-vra` folder.

```
cd vro-node-vra
```

4 Create a Node.js script called `handler.js`.

```
touch handler.js
```

The `handler.js` script must define one function that accepts two arguments, the context of the vRealize Orchestrator workflow run and the bound vRealize Orchestrator inputs.

```
exports.handler = (context, inputs) => {
  console.log('Hello, your inputs were ' + inputs);
  return null;
}
```

Note Using standard logging libraries, everything you log in the action that uses your script is also shown in the workflow log. The inputs and the return of your script must have corresponding input parameters and return types configured in the vRealize Orchestrator Client. For example, the `vRAUrl` input, in your script must have a corresponding input parameter called `vRAUrl` in the vRealize Orchestrator Client. Similarly, if your script returns a string value, the return type configured in the vRealize Orchestrator Client must also be a string type. If your action returns a complex object, you can use `Properties Or Composite Type` return type.

5 Install the Node.js `request` module.

```
npm install request
```

Important Third-party dependency modules must be installed in the root level `node_modules` folder in your main `vro-node-vra` script folder. Do not move or rename this folder.

6 Add the `request` module to the `handler.js` script.

```
const request = require('request');

exports.handler = (context, inputs) => {
  console.log('Hello, your inputs were ' + inputs);
  return null;
}
```

7 Create a GET request to the vRealize Automation Project API.

```
const token = '';
const vRAUrl = '';
```

```
request.get(vRAUrl + '/iaas/api/projects', { 'auth': { 'bearer': token } }, function
(error, response, body) {
  console.log('Got response ' + body);
});
```

8 Define the `token` and `vRAUrl` values.

- a Retrieve the access token by using the vRealize Automation Identity Service API. See [Get Your Access Token for the vRealize Automation API](#).
- b For the `vRAUrl` value, define the script so it uses a vRealize Orchestrator input parameter with the same name.

```
const vRAUrl = inputs.vRAUrl;
```

- c Add the new values to the `handler.js` file.

```
const request = require('request');
exports.handler = (context, inputs, callback) => {
  const vRAUrl = inputs.vRAUrl;
  const token = 'ACCESS_TOKEN';
  request.get(vRAUrl + '/iaas/api/projects', { 'auth': { 'bearer': token } },
function (error, response, body) {
  console.log('Got response ' + body);
  callback(null, JSON.parse(body));
});
}
```

Note Because the response from the vRealize Automation Project API is returned in a JSON format, use a `Properties Of Composite Type` return type for your vRealize Orchestrator action.

- 9 Create a ZIP package that contains the `handler.js` file and `node_modules` folder of your request module.

```
zip -r --exclude=*.zip -X vro-node-vra.zip .
```

What to do next

Import the Node.js script into a vRealize Orchestrator action. See [Create Actions in the vRealize Orchestrator Client](#).

Create a PowerShell Script That Calls the vRealize Automation Project API

Create a sample script that uses PowerShell to call the vRealize Automation Project API.

Procedure

- 1 On your local machine, open a command-line shell.

2 Create a `vro-powershell-vra` folder.

```
mkdir vro-powershell-vra
```

3 Navigate to the `vro-powershell-vra` folder.

```
cd vro-powershell-vra
```

4 Create a PowerShell script called `handler.ps1`.

```
touch handler.ps1
```

The `handler.ps1` script must define one function that accepts two arguments, the context of the vRealize Orchestrator workflow run and the bound vRealize Orchestrator inputs.

```
function Handler {
    Param($context, $inputs)

    $inputsString = $inputs | ConvertTo-Json -Compress
    Write-Host "Inputs were $inputsString"
}
```

Note Using standard logging libraries, everything you log in the action that uses your script is also shown in the workflow log. The inputs and the return of your script must have corresponding input parameters and return types configured in the vRealize Orchestrator Client. For example, the `vRAUrl` input, in your script must have a corresponding input parameter called `vRAUrl` in the vRealize Orchestrator Client. Similarly, if your script returns a string value, the return type configured in the vRealize Orchestrator Client must also be a string type. If your action returns a complex object, you can use `Properties Or Composite Type` return type.

5 Install the PowerShell assert module.

Important Third-party dependency modules must be installed in a root level folder in your main `vro-powershell-vra` script folder. For this use case, you create a `Modules` folder for your assert module.

a Create a `Modules` folder.

```
mkdir Modules
```

b Install the assert module.

```
pwsh -c "Save-Module -Name Assert -Path ./Modules/ -Repository PSGallery"
```

6 Add the assert module to the handler.ps1 script.

```
Import-Module Assert

function Handler {
    Param($context, $inputs)

    $inputsString = $inputs | ConvertTo-Json -Compress
    Write-Host "Inputs were $inputsString"
}
```

7 Create a GET request to the vRealize Automation Project API that uses the Invoke-RestMethod cmdlet.

```
$token = ''
$vRAUrl = ''
$projectsUrl = $vRAUrl + "/project-service/api/projects"
$response = Invoke-RestMethod $projectsUrl + '/iaas/api/projects' -Headers
@{'Authorization' = "Bearer $token"} -Method 'GET'

Write-Host "Got response: $response"
```

8 Define the token and vRAUrl values.

- a Retrieve the access token by using the vRealize Automation Identity Service API. See [Get Your Access Token for the vRealize Automation API](#).
- b Add the `Assert-NotNull` and `Assert-Type` assert module attributes.

```
$token | Assert-NotNull
$token | Assert-Type String
```

- c For the `vRAUrl` value, define the script so it uses a vRealize Orchestrator input parameter with the same name.

```
$vRAUrl = $inputs.vRAUrl
```

- d Add the new values to the `handler.ps1` file.

```
Import-Module Assert
$errorActionPreference = "Stop"
function Handler {
    Param($context, $inputs)
    $token = "ACCESS_TOKEN"
    $token | Assert-NotNull
    $token | Assert-Type String
    $vRAUrl = $inputs.vRAUrl
    $projectsUrl = $vRAUrl + "/project-service/api/projects"
    $response = Invoke-RestMethod $projectsUrl -Headers @{'Authorization' = "Bearer $token"} -Method 'GET'

    Write-Host "Got response: $response"

    return $response
}
```

Note Because the response from the vRealize Automation Project API is returned in a JSON format, use a `Properties` or `Composite` `Type` return type for your vRealize Orchestrator action.

- 9 Create a ZIP package that contains the `handler.ps1` file and `Modules` folder of your `assert` module.

```
zip -r --exclude=*.zip -X vro-powershell-vra.zip .
```

What to do next

Import the PowerShell script into a vRealize Orchestrator action. See [Create Actions in the vRealize Orchestrator Client](#).

Managing Workflows

5

A workflow is a series of actions and decisions that you run sequentially. vRealize Orchestrator provides a library of workflows that perform common management tasks. vRealize Orchestrator also provides libraries of the individual actions that the workflows perform.

Workflows combine actions, decisions, and results that, when performed in a particular order, finish a specific task or a specific process in a virtual environment. Workflows perform tasks such as provisioning virtual machines, backing up, performing regular maintenance, sending emails, performing SSH operations, managing the physical infrastructure, and other general utility operations. Workflows accept inputs according to their function. You can create workflows that run according to defined schedules, or that run if certain anticipated events occur. Information can be provided by you, by other users, by another workflow or action, or by an external process such as a Web service call from an application. Workflows perform some validation and filtering of information before they run.

Workflows can call upon other workflows. For example, you can have workflow that calls up another workflow to create a new virtual machine.

You create workflows by using the vRealize Orchestrator Client interface's integrated development environment (IDE), that provides access to the workflow library and the ability to run workflows on the workflow engine. The workflow engine can also take objects from external libraries that you plug in to vRealize Orchestrator. This feature allows you to customize processes or implement functions that third-party applications provide.

This chapter includes the following topics:

- [Standard Workflows in the vRealize Orchestrator Workflow Library](#)
- [Create Workflows in the vRealize Orchestrator Client](#)
- [Edit Workflows and Actions from the Parent Workflow](#)
- [vRealize Orchestrator Input Form Designer](#)
- [Requests for User Interaction in the vRealize Orchestrator Client](#)
- [Schedule Workflows in the vRealize Orchestrator Client](#)
- [Find Object References in Workflows](#)

Standard Workflows in the vRealize Orchestrator Workflow Library

vRealize Orchestrator provides a standard library of workflows that you can use to automate operations in your virtual infrastructure. The workflows in the standard library are locked in the read-only state. To customize a standard workflow, you must duplicate that workflow. Duplicate workflows or custom workflows that you create are fully editable.

The contents of the workflow library are accessible through the **Library > Workflows** menu of the HTML5-based vRealize Orchestrator Client. Both standard and custom workflows in the client are organized by using tags. For example, you can access the **Generate key pair** workflow by entering **SSH** in the workflow library search box.

Note You cannot add new tags to standard workflows, unless you duplicate the workflow.

Create Workflows in the vRealize Orchestrator Client

You can use the vRealize Orchestrator Client to create and edit workflows.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Select **Library > Workflows**.
- 3 Click **New Workflow**.
- 4 Enter the name of the new workflow and click **Create**.
- 5 Use the workflow editor to configure the variables, workflow inputs and outputs, schema structure, and presentation of the workflow.
- 6 To finish editing the workflow, click **Save**.

Note You can track changes to workflows in the **Version History** tab. For more information, see [vRealize Orchestrator Object Version History](#).

What to do next

You can use the vRealize Orchestrator token replay feature to optimize the performance of your workflows. For more information, see [Using Workflow Token Replay in the vRealize Orchestrator Client](#).

Edit Workflows and Actions from the Parent Workflow

Edit workflows and actions directly from the parent workflow in the vRealize Orchestrator Client.

Editing child workflows and actions directly from the parent workflow can help streamline workflow development.

Prerequisites

Create a workflow that calls up another workflow, action, or both.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Workflows**, and select your workflow.
- 3 Select the **Schema** tab.
- 4 Depending on the object type, double-click the **Workflow element** or **Action element** in the workflow canvas.
- 5 Edit the object.
- 6 To finish editing the child workflow or action, click **Save**.
- 7 To return to the parent workflow, close the object editor.

vRealize Orchestrator Input Form Designer

If a workflow requires input parameters, it opens a dialog box in which users enter the required values. You can organize the content, layout, and presentation of this dialog box with the input form designer.

The input form designer is located in the **Input Form** tab of the workflow editor. This designer consists of a navigation menu, design canvas, and properties menu. You can drag inputs and generic elements from the left menu to the design canvas. In the canvas, you can set the position of the input parameters, organize them into separate input tabs, and configure the input parameter properties.

Note You cannot use content from the **Variables** tab of the workflow editor in the input form designer. You can only use parameters from the **Input/Output** tab.

Generic elements

You can add generic elements, like drop-down menus and password text boxes, to the input form designer. Generic elements do not correspond to actual input parameters, but can be bound to input parameters.

Create the Workflow Input Parameters Dialog Box in the vRealize Orchestrator Client

You can use the input form designer to create and customize the workflow input parameter dialog box.

Prerequisites

Verify that the workflow has a defined list of input parameters.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Workflows**.
- 3 Select your custom workflow.
- 4 Click the **Input Form** tab.
- 5 (Optional) Create tabs for use in the input dialog box.
You can use tabs to organize the structure of your dialog box.
- 6 Select your input parameters.
- 7 Edit the properties of the input parameters.
For more information on input parameter properties, see [Input Parameter Properties in the vRealize Orchestrator Client](#).
- 8 (Optional) Add generic elements to the canvas and bind them to input parameters.
- 9 (Optional) Add external validation to the input parameters. For more information, see [Using Actions to Validate vRealize Orchestrator Workflow Inputs](#).
- 10 Click **Save**.

Results

You created the layout of the workflow dialog box and set the properties of the input parameters.

Input Parameter Properties in the vRealize Orchestrator Client

You can set parameter properties to constrain the input parameters that users provide when they run vRealize Orchestrator workflows.

With vRealize Orchestrator, you can define the parameter properties used to quantify the input parameter values used in workflows. The parameter properties you define impose limits on the types and values of the input parameters that users can provide in vRealize Orchestrator workflows.

Parameter properties validate the input parameters and modify the presentation of the text boxes that appear in the input parameters dialog box. Some parameter properties can create dependencies between parameters.

Parameter Property	Description
Label	Set the input parameter label.
Display type	Set the input text box display type.
Visibility	Set the visibility of the input parameter.
Read-only	Set the input text box as read-only.
Custom help	Set the input parameter signpost description.

Parameter Property	Description
Default value	Set the default value of the input parameter.
Step	Used for number type inputs. Set by how much the value of the input parameter increases per click.
Required	Sets if the input parameter value is mandatory or not.
Regular expression	Validates the input by using a regular expression.
Minimum value	Set the minimum value or length of the parameter.
Maximum value	Set the maximum value or length of the parameter.
Match text box	Set the input parameter value to match the value of another input parameter.
Value source	Set the value source of the parameter properties in the Appearance , Value , and Constraints tabs. Note You can import the value of external actions by using External source . The filtering of available actions is done by parameter type.

Using Actions to Validate vRealize Orchestrator Workflow Inputs

Use external actions to validate the inputs of your custom workflows.

Prerequisites

Create a custom workflow with input parameters. For more information, see [Create Workflows in the vRealize Orchestrator Client](#).

You can use the input form designer to create external validations for your workflow inputs. External validations use action scripts that return a string value when the input parameter value contains an error. If the input parameter value is valid, the external validation returns nothing.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Create a validation action.
 - a Navigate to **Library > Actions**.
 - b Click **New Action**.
 - c Enter the required information on the **Summary** tab.
 - d Enter the validation action input parameters.

Note The names of the validation action input parameters must be identical to the names of the workflow input parameters that are being validated.

- e Enter the script of the validation action on the **Script** tab.

```

if (in_1=="invalid") {
    return "in_1 can't be invalid!";
}
if (in_2=="invalid") {
    return "in_2 can't be invalid!";
}

//inputs are valid, return nothing

```

Note The preceding script is a simple example and does not represent the full scope of the validation scripts that can be used.

- f Click **Save**.
- 3 Apply external validation.
 - a Navigate to **Library > Workflows**.
 - b Select your custom workflow.
 - c Select the **Input Form** tab.
 - d Select the clipboard icon on the top-left of the screen.
 - e Drag an vRealize Orchestrator validation element into the canvas.
 - f Select the validation element, enter a validation label, and select the validation action.
 - g (Optional) Create additional validation elements.
 - h Click **Save**.
 - 4 Run the workflow.

If the validation encounters an error, it returns a string. If the validation is successful, the validation returns nothing and workflow run continues.

Results

You have created an external validation for your custom vRealize Orchestrator workflow.

Requests for User Interaction in the vRealize Orchestrator Client

Workflows can request additional user input before they can finish.

Workflows requiring further user interaction suspend operations until the requested input parameters are provided by the user. Workflows define which users can provide the requested information and send requests for interaction accordingly. Workflows waiting for user input are displayed in the **Recent Workflow Runs** panel of the vRealize Orchestrator Client dashboard and the top-right notification menu.

Schedule Workflows in the vRealize Orchestrator Client

You can use scheduling to automate your vRealize Orchestrator workflow runs.

When you schedule workflow runs, you set the date, time, and intervals at which the scheduled task runs.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Select your workflow from the **Library** menu, and on the workflow panel, click **Schedule**.
- 3 Configure the scheduled task parameters in the **General**, **Scheduling**, and **Workflow** categories.

Note The **Workflow** parameter category is visible only for workflows that require input parameters .

Parameter	Description
Name	The name of the scheduled task.
Description	A short description detailing the purpose of the scheduled task.
Start	The date and time of the first scheduled run of the workflow.
Start if in the past	Select whether to start the workflow, if the scheduled time is in the past. Yes starts the scheduled workflow immediately. No starts the workflow at the next scheduled recurrence.
Schedule	Set the recurrence pattern and event trigger entries of the scheduled task.
End date	Only visible if No Recurrence is selected. Set the date and time of when the scheduled task ends.
Workflow	Enter the input parameters of the workflow.

- 4 Click **Create**.

Results

You have created a scheduled task for the workflow. Scheduled workflows appear under **Activity** > **Scheduled**. You can delete scheduled tasks by clicking **Delete** on the schedule panel.

Edit Scheduled Task in the vRealize Orchestrator Client

Scheduled tasks can be edited to change parameters such as date, time, and recurrence of the scheduled workflow.

Prerequisites

Create a scheduled workflow task.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Select your scheduled task from **Activity > Scheduled**.
- 3 Click **Edit** on the workflow panel.
- 4 Edit the schedule and click **Save**.

Note Input parameters set when creating the scheduled task are read-only and cannot be edited. To change these parameters, create a new scheduled task for this workflow.

Find Object References in Workflows

As a workflow developer, you can use object reference information to optimize your development life cycle.

With the vRealize Orchestrator Client, you can find object reference information. This feature has two functions:

- **Find Dependencies:** find information about object dependencies in your workflows. Dependencies can include other workflows, actions, resource elements, and configuration elements.
- **Find Usages:** learn if the selected workflow is used in other workflows in the vRealize Orchestrator Client library.

You can access information about object references from the workflow editor or from the vRealize Orchestrator Client library in either Card View, List View, or Tree View. For more information on the different types of content organization of the vRealize Orchestrator Client library, see [Content Organization in the vRealize Orchestrator Client](#).

The following procedure demonstrates how you can access referenced objects from the workflow editor.

Prerequisites

Develop a workflow that includes at least one object reference.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Workflows**, and select your workflow.

- 3 To find information about object dependencies, click **Find Dependencies**.

Note While on the dependencies pop-up window, you can select referenced objects from the list. Selecting an object opens a separate vRealize Orchestrator Client tab where you can view the details of the selected object or edit it.

- 4 To find information about where the selected workflow is used, click **Find Usages**.

Managing Actions

6

You can modify your vRealize Orchestrator workflows by adding actions scripts.

The vRealize Orchestrator Client provides libraries of predefined actions and an action editor for custom action scripts. Actions represent individual functions that you use as building blocks in workflows.

Actions are JavaScript functions. Actions can take multiple input parameters and have a single return value. Actions can call on any object in the vRealize Orchestrator API, or objects in any API that you import into vRealize Orchestrator by using a plug-in.

When a workflow runs, an action takes input parameters from the workflow's variables. These variables can be either the workflow's initial input parameters, or variables that other elements in the workflow set when they run.

The action editor includes an autocomplete feature for scripts and an API Explorer featuring the available scripting types and their documentation.

This chapter includes the following topics:

- [Create Actions in the vRealize Orchestrator Client](#)
- [Running and Debugging Actions](#)
- [Core Concepts for Python, Node.js, and PowerShell Scripts](#)
- [Runtime Limits for Python, Node.js, and PowerShell Scripts](#)

Create Actions in the vRealize Orchestrator Client

You can use the vRealize Orchestrator Client to create, edit, and delete action scripts.

Starting with vRealize Orchestrator 8.1, you can use the following runtimes:

- Python 3.7
- Node.js 12
- PowerCLI 11/Powershell 6.2
- PowerCLI 12/Powershell 7

Note The PowerCLI runtime includes PowerShell and the following modules:
VMware.PowerCLI, PowerNSX, PowervRA.

Prerequisites

Before creating a Python, Node.js, or PowerShell script, verify that you are familiar with the core concepts for developing vRealize Orchestrator compatible scripts that use these runtimes. See [Core Concepts for Python, Node.js, and PowerShell Scripts](#) .

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Actions**.
- 3 Click **New Action**.
- 4 On the **General** tab, enter the name and module name of the action.

Note The name and module name must be unique for every action. The action name must be a valid JavaScript function. The action name must be a single word that can only contain letters, numbers, and the dollar ("\$\$") and underscore ("_") symbols. The module name must consist of words separated by the dot (".") character.

- 5 (Optional) Create a description, version number, tags, and group permissions for the action.
- 6 On the **Script** tab, add action inputs, select the return type of the output, and write the script.

Note By selecting **Zip** from the **Type** drop-down menu, you can import an external script source and, if applicable, its dependency modules.

- 7 To finish editing the action, click **Save**.

A message states that the action is saved.

What to do next

To view a use case example of how you can use vRealize Orchestrator actions, see [How to Integrate Amazon Web Services in vRealize Orchestrator by Using Python](#).

Running and Debugging Actions

You can improve your actions by running and debugging them directly from the action editor.

Starting with vRealize Orchestrator 8.1, you can run and debug actions directly from the action editor of the vRealize Orchestrator Client. With this feature, you can guarantee that your actions perform as expected when they are integrated into your workflows.

Run Actions in the vRealize Orchestrator Client

As a workflow designer, you want to run your actions before integrating them into a workflow.

Prerequisites

Create an action. See [Create Actions in the vRealize Orchestrator Client](#).

Procedure

- 1 Log in the vRealize Orchestrator Client.
- 2 Navigate to **Library > Actions**, and select the action you want to run.
- 3 Click **Run**.
- 4 Enter the required input parameters, and click **Run**.

After the action run finishes, click the **Results/Inputs** tab. If the action run encountered an error, it is displayed on this tab in a red color. You can view the details of the action run from the **Action Results** element.

Note The results of the action run are not saved.

Debug Actions in the vRealize Orchestrator Client

As a workflow designer, you can debug actions by inserting breakpoints into your script.

vRealize Orchestrator includes a built-in debugging tool that you can use to debug the script and input properties of your action. The debug process can be initiated in the action editor by inserting breakpoints into the script lines of your action.

Note The built-in debugging tool only works with actions that use the default JavaScript runtime. For an example of how you can debug action scripts that use different runtimes, see [Debug the Amazon Web Services Action](#).

Prerequisites

Create an action. See [Create Actions in the vRealize Orchestrator Client](#).

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Actions**, and select the action you want to debug.
- 3 In the action editor, add breakpoints to the lines of your action script you want to debug.
- 4 Click **Debug**.
- 5 Enter the input parameters of your action, and click **Run**.

An action run in debug mode begins.

- 6 When the action run is paused after reaching a breakpoint, select one of the following options:

Option	Description
Continue	Resumes the action run until another breakpoint is reached or the action run finishes.
Step into	Step into the current action function. If the debugger cannot go deeper into the current line of the function, it performs a Step over operation.

Option	Description
Step over	The debugger continues into the next line of the current function.
Step return	The debugger goes into the line that will perform when the current function returns.

- 7 (Optional) On the **Debugger** tab, add expressions.
- 8 (Optional) On the **Debugger** tab, edit the value of your variables.

Core Concepts for Python, Node.js, and PowerShell Scripts

When creating your script for use in vRealize Orchestrator, you must verify that your script has the correct structure and formatting.

Supported Runtimes

For developing vRealize Orchestrator actions and workflows, you can use the following runtimes:

- Python 3.7
- Node.js 12
- PowerCLI 11/Powershell 6.2
- PowerCLI 12/Powershell 7

Note The PowerCLI runtime includes PowerShell and the following modules: VMware.PowerCLI, PowerNSX, PowervRA.

You can add any custom source code to the new runtimes, but to accept context and inputs, and return a result from and to the vRealize Orchestrator engine, you must follow the correct functional format.

Scripting Recommendations

For simpler scripting tasks, you can add **Scriptable task** elements to your workflow schema. You can use vRealize Orchestrator actions for more complex scripting tasks.

Using actions provides two specific benefits:

- Actions can be created, updated, imported, and exported independently from workflows.
- Actions are standalone objects that can be run and debugged in their own environment which can lead to a smoother development process. See [Running and Debugging Actions](#) .

Script Function Requirements

The default name for your script function is **handler**. The function accepts two arguments, context and input. Context is a map object, containing system information. For example, `vroURL` can contain the URL of the vRealize Orchestrator instance you want to call, while `executionId` contains the token ID of a workflow run.

An input is a map object containing all inputs that are provided to the actions. For example, if you define an input in your action called `myInput`, you can access it from the `inputs` argument, such as `inputs.myInput` or `inputs["myInput"]`, depending on your runtime. Anything that you return from the function, is the result of the action. Therefore, the return type of your action must correspond to the type of content that the script returns in vRealize Orchestrator. If you return a primitive number, the action return type must be a number type. If you return a string, the action return type must be a string type. If you return a complex object, the return type must be mapped to either `Properties` or `Composite Type`. These same principles also apply to arrays.

Supported input and output parameter types for Python, Node.js, and PowerShell runtimes:

- `String`
- `Number`
- `Boolean`
- `Date`
- `Properties`
- `Composite Type`

Define the Entry Handler

By default, the value of the entry handler is `handler.handler`. This value means that the vRealize Orchestrator engine looks for a top-level file in your ZIP package called `handler.py`, `handler.js`, or `handler.ps1`, that includes a function called `handler`. Any differences to the names of the function and handler file must be reflected in the value of the entry handler. For example, if your main handler is called `index.js` and your function is called `callMe`, you must set the value for the entry handler to `index.callMe`.

Debug Runtime Scripts in an External IDE

vRealize Orchestrator supports debugging Python and Node.js scripts in an external IDE. You cannot debug PowerShell scripts in an external IDE.

Runtime Limits for Python, Node.js, and PowerShell Scripts

Some Python, Node.js, or PowerShell scripts can require you to change the memory and timeout values in the vRealize Orchestrator Client.

The vRealize Orchestrator Client uses a set of default memory and timeout values for Python, Node.js, and PowerShell action scripts:

- Memory: 64 MB
- Timeout: 180 seconds

If your action script exceeds one or both of these default values, the action run fails. For example, your action script might use multiple third-party dependency modules. In such a scenario, the default memory limit of 64 MB might not be enough.

To avoid failed action runs due to insufficient resources, change the memory and timeout values from the action editor.

Note You can also consider breaking up your script into multiple scriptable task elements, that can be added to your workflows.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Actions**, and select your action.
- 3 Select the **Script** tab.
- 4 Under **Runtime limits**, change the memory and timeout values.
- 5 Click **Save**.
- 6 To test the new runtime limits, click **Debug**.

Managing Configuration Elements

7

A configuration element is a list of variables, that you can use to configure constants across a whole vRealize Orchestrator server deployment.

You can use configuration elements to make variables available to all the workflows, actions, and policies running on the vRealize Orchestrator server.

If you create a package containing a workflow, action, or policy that uses a variable from a configuration element, vRealize Orchestrator automatically includes the configuration element in the package. If you import a package containing a configuration element into another vRealize Orchestrator server, you can import the configuration element variable values as well. For example, if you create a workflow that requires variable values that depend on the vRealize Orchestrator server on which it runs, setting those variables in a configuration element lets you export that workflow, so that another vRealize Orchestrator server can use it. Configuration elements therefore allow you to exchange workflows, actions, and policies between servers more easily.

Note You cannot import values of a configuration element variable from a configuration element exported from vRealize Orchestrator 5.1 or earlier.

This chapter includes the following topics:

- [Create Configuration Elements in the vRealize Orchestrator Client](#)

Create Configuration Elements in the vRealize Orchestrator Client

With configuration elements, you can set common variables across an vRealize Orchestrator server. All elements that are running in the server can use the variables you set in a configuration element.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Assets > Configurations**.
- 3 Select **New Configuration**.
- 4 Enter the configuration element name.

- 5 Select the **Variables** tab.
- 6 To create a local variable, click **New**.
 - a Enter the variable name.
 - b Select the variable type.

Note To create an array of configuration variables, select the **Array** check box.

- c (Optional) Enter a value for the configuration variable.
 - d Click **Save**.
- 7 To finish creating a configuration element, click **Save**.

What to do next

You can use the configuration element to provide variables to workflows, actions, or policies.

Managing Policies



Policies are event triggers that monitor the activity of the system. Policies respond to predefined events issued by changes in the status or performance of specific vRealize Orchestrator objects.

Policies are a series of rules, gauges, thresholds, and event filters that run certain workflows or scripts when specific predefined events occur in vRealize Orchestrator or in the technologies that vRealize Orchestrator accesses through plug-ins. vRealize Orchestrator constantly evaluates the policy rules while the policy is running. For instance, you can implement policy gauges and thresholds that monitor the behavior of vCenter Server objects of the `VC:HostSystem` and `VC:VirtualMachine` types.

This chapter includes the following topics:

- [Create and Apply Policies in the vRealize Orchestrator Client](#)
- [Policy Elements in the vRealize Orchestrator Client](#)
- [Manage Policy Runs in the vRealize Orchestrator Client](#)

Create and Apply Policies in the vRealize Orchestrator Client

You can use policies to monitor the activity of the vRealize Orchestrator system for specific events.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Library > Policies**.
- 3 Select **New Policy**.
You have created a blank policy.
- 4 Enter a policy name and version number.
- 5 Select the **Variables** tab.
- 6 To create a local variable, click **New**.
 - a Enter the variable name.
 - b Select the variable type.

Note To create an array of policy variables, select the **Array** check box.

- c Enter the variable value.

Note To import the value of a configuration element variable, you can use **Bind to configuration**.

- d Click **Save**.

- 7 On the **Definition** tab, add policy elements and set event handlers.

For more information on policy elements, see [Policy Elements in the vRealize Orchestrator Client](#).

- 8 Click **Save**.

You have configured the policy.

What to do next

To start a policy, select the policy and click **Run**. Enter the policy run name and, if prompted, the required input parameters.

To view the policy status, navigate to **Activity > Policy Runs**.

Policy Elements in the vRealize Orchestrator Client

You can use policy elements to run predefined vRealize Orchestrator workflows or scripts when an event occurs.

You can add a policy element to trigger workflow or script runs as a response to events triggered by objects. With the periodic event element, you can schedule workflow or script runs. With the root element, you can set the start or stop behavior of policies. Policy elements can have event handlers that define when policy elements must run.

Note Event handlers that activate policy elements can be either workflows or action scripts. If you add both a workflow and a script to an event handler, the policy ignores the script trigger and only uses the workflow trigger.

Event Handler	Description
OnInit	The policy element is triggered every time you start the policy.
OnExit	The policy element is triggered every time you stop the policy.
OnExecute	Used by the periodic event element. Triggers the policy element during the time specified in the periodic event element.

Note Technologies plugged in to the vRealize Orchestrator database can possess unique event handlers. For example, through the SNMP plug-in, you can use the **OnTrap** event handler when creating SNMP-based policy elements.

Policy elements are configured on the **Definition** tab of the policy edit window.

Manage Policy Runs in the vRealize Orchestrator Client

You can use the vRealize Orchestrator Client to manage the policy priority and server start-up behavior of policies for when the vRealize Orchestrator server is restarted.

Prerequisites

Create and run a policy. For more information, see [Create and Apply Policies in the vRealize Orchestrator Client](#).

Procedure

- 1 Log in to the vRealize Orchestrator Client as an administrator.
- 2 Navigate to **Activity > Policy Runs**.
- 3 Click the policy run you want to manage.
- 4 Click **Stop**.

The policy state changes to **Stopped**.

- 5 On the **General** tab, set the policy priority and server start-up behavior.
- 6 To restart the policy, click **Run**.

The policy state changes to **Running**.

Managing Resource Elements

9

Workflows can use objects that you create independently of vRealize Orchestrator as attributes. To use external objects as attributes in workflows, you import them into the server as resource elements.

Objects that vRealize Orchestrator workflows can use as resource elements include image files, scripts, XML templates, HTML files, and so on. Any workflows that run in the vRealize Orchestrator server can use any resource elements that you import into vRealize Orchestrator.

After you import an object into vRealize Orchestrator as a resource element, you can make changes to the object in a single location, and propagate those changes automatically to all the workflows that use this resource element.

The maximum size for a resource element is 16 MB.

You can import, export, restore, update, and delete a resource element.

Managing Packages

10

Use the vRealize Orchestrator Client to create, export, and import packages. Packages can be used to export workflow objects for use on other vRealize Orchestrator instances.

Packages can contain workflows, actions, policies, configuration elements, or resources elements.

When you add an element to a package, vRealize Orchestrator checks for dependencies and adds any dependent elements to the package. For example, if you add a workflow that uses actions or other workflows, vRealize Orchestrator adds those actions and workflows to the package.

When you import a package, the server compares the versions of the different elements of its contents to matching local elements. The comparison shows the differences in versions between the local and imported elements. The user can decide whether to import the package, or can select specific elements to import.

For most objects created in the vRealize Orchestrator Client, aside from resource elements, packages are the only way to export and import these objects.

Packages use digital rights management to control how the receiving server can use the contents of the package. vRealize Orchestrator signs packages and encrypts the packages for data protection. Packages can track which users export and redistribute elements by using X509 certificates.

Create a Package in the vRealize Orchestrator Client

You can export and import workflows, policies, actions, plug-in references, resource elements, and configuration elements in packages. All dependent elements related to package objects are added to the package automatically, to ensure compatibility between versions. To delete dependent elements, you must first remove the related package object.

For most objects created in the vRealize Orchestrator Client, aside from resource elements, packages are the only way to export and import these objects.

Prerequisites

Verify that the vRealize Orchestrator server contains objects like workflows, actions, and policies, that you can add to a package.

Procedure

- 1 Log in to the vRealize Orchestrator Client.

- 2 Navigate to **Assets > Packages**.
- 3 Click **New Package**.
- 4 On the **General** tab, enter a name and description for the package.

Note You cannot use special characters when naming packages in the vRealize Orchestrator Client.

- 5 On the **Content** tab, click **Add**.
- 6 Select the objects that you want to add to the package and click **Add**.

Note Dependent elements are added to the package automatically, but are not displayed in the **Content** tab during package creation. To view dependent elements, select the **Content** tab after package creation.

- 7 To finish creating the package, click **Create**.

Export a Package in the vRealize Orchestrator Client

You can use the vRealize Orchestrator Client to export packages to another vRealize Orchestrator environment.

Prerequisites

Create a package containing the vRealize Orchestrator objects you want to export. For more information, see [Create a Package in the vRealize Orchestrator Client](#).

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Assets > Packages**.
- 3 Click **Export** on the package.
- 4 (Optional) Select additional export options.

Option	Description
Add configuration attribute values to package	Export the attribute values of the configuration elements.
Add configuration SecureString attribute values to package	Export the <code>SecureString</code> configuration attribute values.
Add global tags to package	Export the global tags.

- 5 Set the access rights for users who import the package.

Option	Description
View contents	The user can view the package content.
Add to package	The user can add content from the imported package to other packages.
Edit contents	The user can edit the package content.

- 6 Click **Ok**.

Note Files with the `.package` extension are saved to a default folder on your local machine. To set a custom folder, you can change the storage settings in your browser.

Results

You exported the package. You can now use the exported objects on another vRealize Orchestrator environment.

Import a Package in the vRealize Orchestrator Client

Use the vRealize Orchestrator Client to import workflow packages. By importing packages, you can reuse objects from one vRealize Orchestrator server on another server.

Prerequisites

- Back up any standard vRealize Orchestrator objects that you have modified.
- On the remote server, create and export a package with the objects you want to import.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Assets > Packages**.
- 3 Click **Import**, browse to the `.package` file that you want to import, and click **Open**.
- 4 Review the imported package information.
 - a The **General** tab contains information about the imported package like the name, description, number of contained items, and certificate information.
You might be prompted to indicate that you trust the publisher certificate of the source vRealize Orchestrator instance before you can import the file.
 - b The **Package elements** tab lists the objects included in the import file. If the version of an object in the package is later than the version on the server, the system selects that object version for import. Earlier versions of vRealize Orchestrator elements must be selected manually.

- c Deselect **Import Configuration Attribute Values** if you do not want to import the attribute values of the configuration elements from the package.
 - d From the drop-down menu, select whether you want to import tags.
- 5 Click **Import**.

Troubleshooting in the vRealize Orchestrator Client

11

You can troubleshoot and monitor your vRealize Orchestrator instance by using metrics, token replay, validation, and debugging.

This chapter includes the following topics:

- [Metric Data in the vRealize Orchestrator Client](#)
- [Using Workflow Token Replay in the vRealize Orchestrator Client](#)
- [Validating vRealize Orchestrator Workflows](#)
- [Debug Workflow Scripts in the vRealize Orchestrator Client](#)
- [Debug Workflows by Schema Element](#)
- [Configuring a Photon OS Container for Python Packages](#)

Metric Data in the vRealize Orchestrator Client

vRealize Orchestrator administrators can use workflow profiling and the System Dashboard metrics to troubleshoot the vRealize Orchestrator system and workflows.

The profiling feature gathers metric data about workflow runs. Workflow profiling is enabled by default. You can disable automatic profiling in **Control Center > Extension Properties > profiler-8.3.0**.

The other source for metric data in the vRealize Orchestrator Client is the System Dashboard, that provides system level metrics. For more information, see [Using the vRealize Orchestrator System Dashboard](#).

Profile Workflows in the vRealize Orchestrator Client

You can profile your workflow runs to troubleshoot and optimize your vRealize Orchestrator environment.

You can use the profiling feature of the vRealize Orchestrator Client to gather useful metric data about your workflow runs. This data can be used to optimize the performance of your workflows. By default, workflow runs are profiled automatically. You can disable automatic profiling from the **Extension Properties** page of the vRealize Orchestrator Control Center and run the profiler manually. To do a manual profiling run, find your workflow in the library and select **Actions > Profile**.

Prerequisites

Run a workflow.

Procedure

- 1 Log in to the vRealize Orchestrator Client.
- 2 Navigate to **Activity > Workflow Runs**.
- 3 Select a workflow run.

On the workflow run schema, you can see data about the individual workflow items. Data includes total run duration, max duration, and number of item runs. You can filter this information from the drop-down menu on the top right of the page.

- 4 Select the **Performance** tab.

This tab provides you with metric data on workflow run CPU times, run duration, token size, and workflow item data.

Note If the workflow run is suspended, for example when the workflow is waiting for further input, the CPU times metric only captures the runtime thread that occurred before completion.

What to do next

Use the data gathered from profiling to optimize your workflow.

Using the vRealize Orchestrator System Dashboard

As an administrator, you can use the vRealize Orchestrator Client System Dashboard to gather useful metric data about the nodes of your vRealize Orchestrator environment.

You can access the System Dashboard from by clicking the **System** tab, on the top of vRealize Orchestrator Client dashboard page. Provided data includes:

- Node status
- Node properties
- Cluster settings. You can only view the cluster settings from the System Dashboard. To change these settings, go to the **Orchestrator Cluster Management** page of the vRealize Orchestrator Control Center.
- Threads info
- Heap memory
- Non-heap memory
- File system use
- Authentication data
- Orchestrator database connection pool
- Process input arguments

This data can be used to monitor the state of individual nodes of your vRealize Orchestrator environment and troubleshoot problems. To navigate between individual nodes, click the tab associated with a node on the top of the System Dashboard.

Using Workflow Token Replay in the vRealize Orchestrator Client

You can use the token replay feature to view the transitions between items in workflow runs.

The token replay feature records contextual information for each transition between workflow items. For each workflow item, token replay records when the workflow run started, ended, and what variables were changed at the end of the workflow item run. Token replay also references the generated script log messages for each workflow item.

Note Data about workflow item transitions is stored in the vRealize Orchestrator PostgreSQL database. This data is removed from the database when the workflow run is deleted.

Prerequisites

- Enable the token replay feature from the Control Center.
 - a Log in to the Control Center as **root**.
 - b Select **Extension Properties**.
 - c Click **tokenreplay-8.3.0**.
 - d To enable the token replay feature, click **Enable**.
 - e Click **Save**.

Note It can take up to 5 minutes for the vRealize Orchestrator server to refresh the extension.

- Run a workflow.

Note By default, the token replay does not run automatically for all workflow runs on your vRealize Orchestrator server. You can run token replays for each workflow individually, or enable the token replay extension for all workflows from the **Extension Properties** page of the Control Center.

Procedure

- 1 (Optional) Enable token replay for all workflow runs on your vRealize Orchestrator server.

Note To run individual token replays without enabling the feature from the Control Center, click **Run with replay** on the workflow editor page.

- a Log in to the Control Center as **root**.
- b Select **Extension Properties**.

- c Click **tokenreplay-8.3.0**.
- d To enable the token replay feature for all workflows, verify that **Record replay for all workflow runs** is enabled.
- e Click **Save**.

Note It can take up to 5 minutes for the vRealize Orchestrator server to refresh the extension.

- 2 Log in to the vRealize Orchestrator Client as an administrator.
- 3 Navigate to **Activity > Workflow Runs**.
- 4 Select a workflow run.
- 5 Select a workflow run item from the left menu.

The **Variable** and **Logs** tabs now display information specific for that workflow item.

Validating vRealize Orchestrator Workflows

vRealize Orchestrator provides a workflow validation tool. Validating a workflow helps identify errors in the workflow and checks that the data flows from one element to the next correctly.

By default, vRealize Orchestrator always performs a workflow validation when you run a workflow.

When you validate a workflow, the validation tool creates a list of any errors or warnings. Clicking an error in the list highlights the workflow element that contains the error.

If you run the validation tool in the workflow editor, the tool provides suggested quick fixes for the errors it detects. Some quick fixes require additional information or input parameters. Other quick fixes resolve the error for you.

Workflow validation checks the data bindings and connections between elements. Workflow validation does not check the data processing that each element in the workflow performs. As a result, a valid workflow might run incorrectly and produce erroneous results if a function in a schema element is incorrect.

Validate a Workflow and Fix Validation Errors in the vRealize Orchestrator Client

You must validate a workflow before you can run it. You can only fix validation errors if you have opened the workflow for editing.

Prerequisites

Verify that you have a complete workflow to validate, with schema elements linked and bindings defined.

Procedure

- 1 Log in to the vRealize Orchestrator Client as an administrator.

- 2 Navigate to **Library > Workflows** and select the workflow you want to validate.
- 3 Click **Edit**.
- 4 Click **Validate** from the top menu.

If the workflow is valid, a confirmation message appears. If the workflow is invalid, a list of errors appears.

- 5 For an invalid workflow, click an error message and take appropriate steps to resolve the problem.

The validation tool highlights the schema element in which the error occurs by adding a red icon to it. Where possible, the validation tool displays a quick fix action.

- If you agree with the proposed quick fix action, click it to perform that action.
- If you disagree with the proposed quick fix action, close the Workflow Validation dialog box and fix the schema element manually.

Important Always check that the fix that vRealize Orchestrator proposes is appropriate.

For example, the proposed action might be to delete an unused attribute, when in fact that attribute might not be bound correctly.

- 6 Repeat the preceding steps until you have eliminated all validation errors.

Results

You validated a workflow and fixed the validation errors.

What to do next

You can run the workflow.

Debug Workflow Scripts in the vRealize Orchestrator Client

You can debug workflow runs by inserting breakpoints in the script of workflow items.

When a breakpoint is reached, you have several options to continue the debugging process.

When you debug an element from the workflow schema, you can view the general information about the workflow run, modify the workflow variables, add expressions to watch, and view log messages.

Note Perform all script debugging in a non-production environment.

Procedure

- 1 Log in to the vRealize Orchestrator Client as an administrator.
- 2 Select a workflow from the library.
- 3 Open the workflow schema, select a workflow element, and click the **Scripting** tab.

- To insert a breakpoint, click the red circle to the left of the line number.

Note You can only insert breakpoints in workflow elements with scripting.

- To run the workflow in the debugging mode, click **Debug**.
If the workflow requires input parameters, you must provide them.
- When the workflow run is paused after reaching a breakpoint, select one of the available options.

Option	Description
Continue	Resumes the workflow run until another breakpoint is reached or the workflow run finishes.
Step into	You can use this option to step into a workflow element. You cannot step into a nested workflow element when you debug a workflow in the workflow editor.
Step over	Skips the current element in the schema and pauses the workflow run on the next element.

Note You can instruct the debugger to ignore the current breakpoint by clicking the breakpoint. This changes the breakpoint symbol to a green triangle.

- (Optional) On the **Debugger** tab, insert expressions to watch.
You can use expressions to follow the completion of specific variables.
- (Optional) On the **Debugger** tab, modify the values of variables.

Debug Workflows by Schema Element

As a workflow designer, you can debug individual schema elements.

Procedure

- Log in to the vRealize Orchestrator Client.
- Navigate to **Library > Workflows**, and select your workflow.
- Select the **Schema** tab.
- Select the workflow element you want to debug, and click the debug button on the top-left of the element.

Note By adding a breakpoint to a **Workflow Element** schema element, you can debug child workflows directly from the parent workflow. When the debugger reaches the **Workflow Element** schema element, it opens the schema view of the child workflow.

- Repeat for any other schema elements you want to debug.
- Click **Debug**.

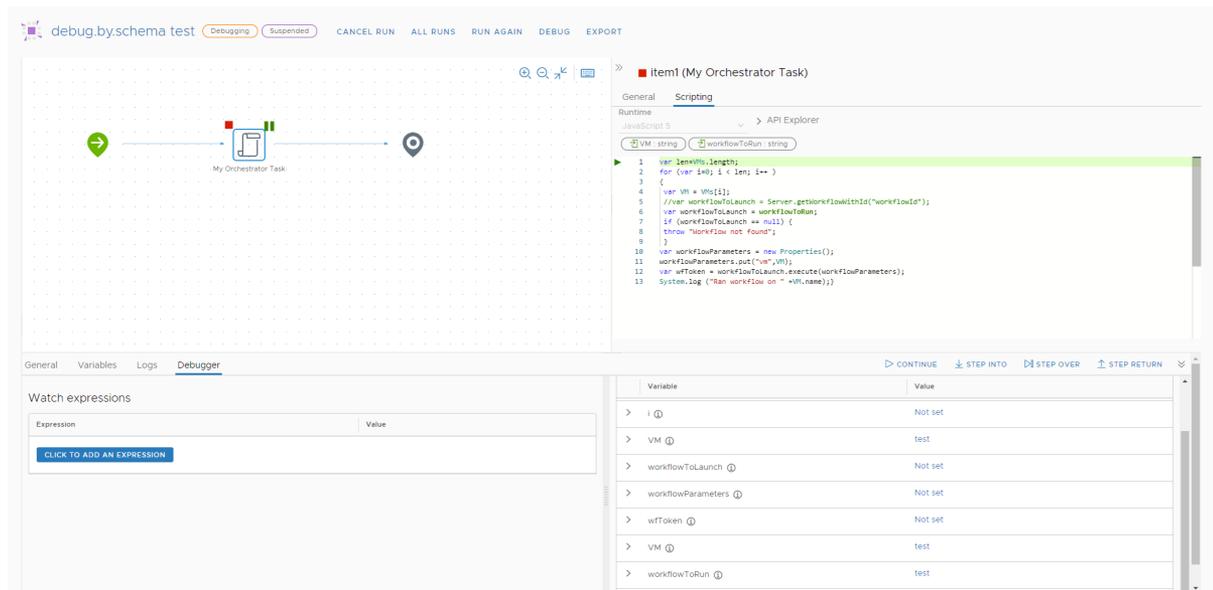
- 7 Enter the requested input parameter values, and click **Run**.

The workflow run begins, and is suspended when the debugger reaches a schema element with a breakpoint.

- 8 When at a breakpoint, select one of the following options:

Option	Description
Continue	Resumes the workflow run until another breakpoint is reached or the workflow run finishes.
Step into	Step into the current workflow function. If the debugger cannot go deeper into the current line of the function, it performs a Step over operation.
Step over	The debugger continues into the next line of the current function.
Step return	The debugger goes into the line that will perform when the current function returns.

- 9 (Optional) On the **Variables** tab, edit the value of your workflow variables.



Configuring a Photon OS Container for Python Packages

Depending on the operating system (OS) used for compiling your Python script, your workflows or actions can fail after importing the relevant ZIP archive to the vRealize Orchestrator Client.

The OS of the runtime container used for Python in vRealize Orchestrator is based on Photon 3.0. Python script packages compiled for another OS, such as Linux for example, are incompatible with the runtime container. This problem can cause the Python script to fail, when you attempt to use it as part of your vRealize Orchestrator workflows or actions. In such a scenario, you receive the following error message in your logs:

```
-04:00errorCannot find module action
```

To resolve this problem, you must install the required Python package in a Photon OS container folder.

Prerequisites

Install Docker. See [Get Docker](#).

Procedure

- 1 Navigate to the parent folder of your Python script.
- 2 Create a container with the base Photon image by mounting a container folder to your parent folder.

Note The following script is a singular Docker command that you must run in its entirety to create a suitable container.

```
docker run -ti -v
$(pwd)/<name_of_folder_that_contains_your_python_script>:/: /
<name_of_folder_that_contains_your_python_script>
photon:3.0
```

- 3 Install Python in the container.

```
tdnf install -y python3-3.7.5-5.ph3 python3-pip-3.7.5-5.ph3
```

- 4 Navigate to the container folder that includes your Python script.
- 5 Add your Python script and packages.

Note Install packages required for your Python script in the `lib` folder.

```
pip3 install <package_name> -t lib/
```

- 6 Exit the container and navigate to the local folder you mounted to the container.
- 7 Compress all relevant files and folders into a ZIP archive.
- 8 Import the ZIP archive into the vRealize Orchestrator Client and validate the script by running it as part of an action.