



Manage and protect apps

Astra Control Service

NetApp

December 16, 2021

Table of Contents

- Manage and protect apps. 1
 - Start managing apps 1
 - Protect apps with snapshots and backups. 9
 - Restore apps 14
 - Clone and migrate apps 15
 - Manage app execution hooks 17

Manage and protect apps

Start managing apps

After you [add a Kubernetes cluster to Astra Control](#), you can install apps on the cluster (outside of Astra Control), and then go to the Apps page in Astra Control to start managing the apps.

App management requirements

Consider the following Astra Control app management requirements:

- **Licensing:** To manage apps using Astra Control Center, you need an Astra Control Center license.
- **Namespaces:** Astra Control requires that an app not span more than a single namespace, but a namespace can contain more than one app.
- **StorageClass:** If you install an app with a StorageClass explicitly set and you need to clone the app, the target cluster for the clone operation must have the originally specified StorageClass. Cloning an application with an explicitly set StorageClass to a cluster that does not have the same StorageClass will fail.
- **Kubernetes resources:** Apps that use Kubernetes Resources not collected by Astra Control might not have full app data management capabilities. Astra Control collects the following Kubernetes resources:
 - ClusterRole
 - ClusterRoleBinding
 - ConfigMap
 - CustomResourceDefinition
 - CustomResource
 - DaemonSet
 - Deployment
 - DeploymentConfig
 - Ingress
 - MutatingWebhook
 - PersistentVolumeClaim
 - Pod
 - ReplicaSet
 - RoleBinding
 - Role
 - Route
 - Secret
 - Service
 - ServiceAccount
 - StatefulSet
 - ValidatingWebhook

Supported app installation methods

You can use the following methods to install apps that you want to manage with Astra Control:

- **Manifest file:** Astra Control supports apps installed from a manifest file using kubectl. For example:

```
kubectl apply -f myapp.yaml
```

- **Helm 3:** If you use Helm to install apps, Astra Control requires Helm version 3. Managing and cloning apps installed with Helm 3 (or upgraded from Helm 2 to Helm 3) are fully supported. Managing apps installed with Helm 2 is not supported.
- **Operator-deployed apps:** Astra Control supports apps installed with namespace-scoped operators. These operators are generally designed with a "pass-by-value" rather than "pass-by-reference" architecture. The following are some operator apps that follow these patterns:
 - [Apache K8ssandra](#)
 - [Jenkins CI](#)
 - [Percona XtraDB Cluster](#)

Note that Astra Control might not be able to clone an operator that is designed with a "pass-by-reference" architecture (for example, the CockroachDB operator). During these types of cloning operations, the cloned operator attempts to reference Kubernetes secrets from the source operator despite having its own new secret as part of the cloning process. The clone operation might fail because Astra Control is unaware of the Kubernetes secrets in the source operator.



An operator and the app it installs must use the same namespace; you might need to modify the deployment .yaml file for the operator to ensure this is the case.

Install apps on your cluster

Now that you've added your cluster to Astra Control, you can install apps on the cluster. Persistent volumes will be provisioned on the new storage classes by default. After the pods are online, you can manage the app with Astra Control.

Astra Control will manage stateful apps only if the storage is on a storage class installed by Astra Control.

- [Learn about storage classes for GKE clusters](#)
- [Learn about storage classes for AKS clusters](#)

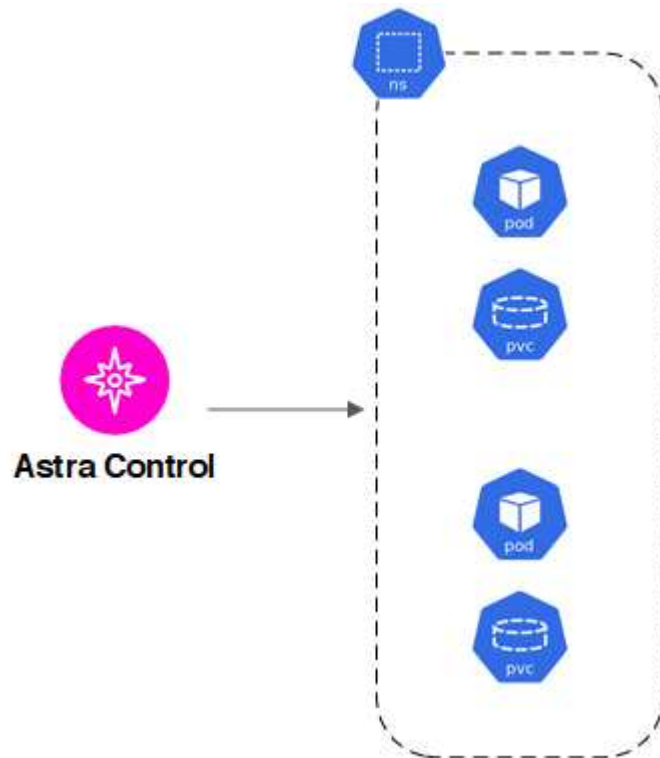
For help with deploying common applications from Helm charts, refer to the following:

- [Deploy MariaDB from a Helm chart](#)
- [Deploy MySQL from a Helm chart](#)
- [Deploy Postgres from a Helm chart](#)
- [Deploy Jenkins from a Helm chart](#)

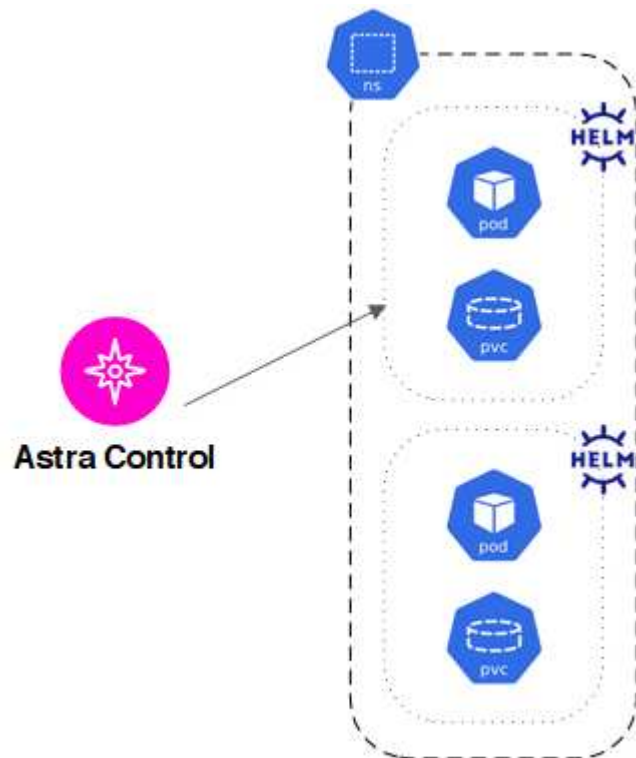
Manage apps

When Astra Control discovers the apps running on your clusters, they are unmanaged until you choose how you want to manage them. A managed application in Astra Control can be any of the following:

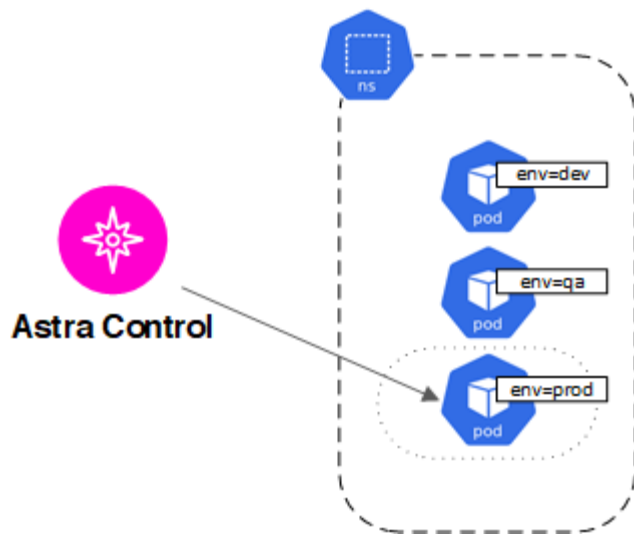
- A namespace, including all resources in that namespace



- An individual application deployed with helm3 within a namespace



- A group of resources that are identified by a Kubernetes label (this is called a *custom app* in Astra Control)



The sections below describe how to manage your apps using these options.

Manage apps by namespace

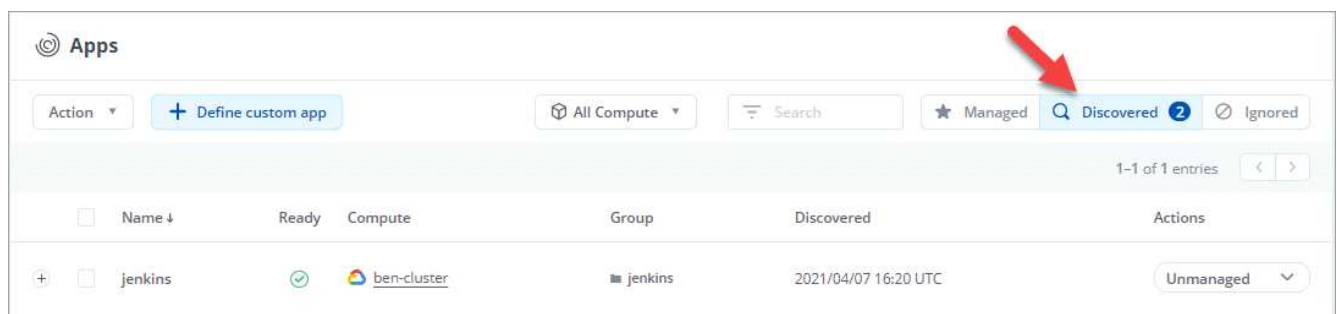
The **Discovered** section of the Apps page shows namespaces and the Helm-installed apps or custom-labeled apps in those namespaces. You can choose to manage each app individually or at the namespace level. It all comes down to the level of granularity that you need for data protection operations.

For example, you might want to set a backup policy for "maria" that has a weekly cadence, but you might need to back up "mariadb" (which is in the same namespace) more frequently than that. Based on those needs, you would need to manage the apps separately and not under a single namespace.

While Astra Control allows you to separately manage both levels of the hierarchy (the namespace and the apps in that namespace), the best practice is to choose one or the other. Actions that you take in Astra Control can fail if the actions take place at the same time at both the namespace and app level.

Steps

1. Select **Applications** and then select **Discovered**.



2. View the list of discovered namespaces and expand a namespace to view the apps and associated resources.

Astra Control shows you Helm apps and custom-labeled apps in namespace. If Helm labels are available, they're designated with a tag icon.

Here's an example with two apps in a namespace:

The screenshot shows the 'Apps' management interface. At the top, there's a header with 'Apps' and a search bar. Below the header, there are filters for 'All Compute', 'Managed', 'Discovered' (3 items), and 'Ignored'. The main table has columns: Name, Ready, Compute, Group, Discovered, and Actions. The table lists three entries: 'jenkins', 'jenkins-jenkins', and 'jenkins2-jenkins'. Each entry has a 'Ready' status (green checkmark), a 'Compute' resource (ben-cluster), a 'Group' (jenkins), and a 'Discovered' timestamp. The 'Actions' column shows a dropdown menu with 'Unmanaged' selected.

Name	Ready	Compute	Group	Discovered	Actions
jenkins	✓	ben-cluster	jenkins	2021/04/07 16:20 UTC	Unmanaged
jenkins-jenkins	✓	ben-cluster	jenkins app.kubernetes.io/name: jenkins +1	2021/04/07 16:20 UTC	Unmanaged
jenkins2-jenkins	✓	ben-cluster	jenkins app.kubernetes.io/name: jenkins +1	2021/04/07 19:26 UTC	Unmanaged

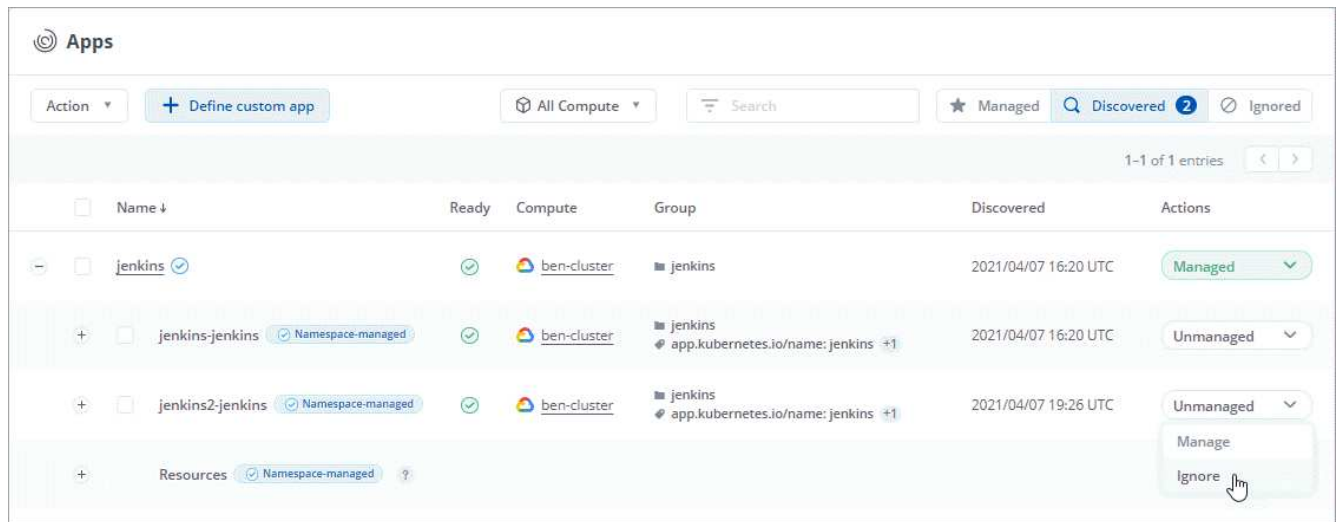
- Decide whether you want to manage each app individually or at the namespace level.
- At the desired level in the hierarchy, select the drop-down list in the **Actions** column and select **Manage**.

The screenshot shows the 'Apps' management interface with the 'Actions' dropdown menu open for the 'jenkins' entry. The dropdown menu has three options: 'Unmanaged', 'Manage', and 'Ignore'. A mouse cursor is pointing at the 'Manage' option. The table structure is the same as the previous screenshot.

Name	Ready	Compute	Group	Discovered	Actions
jenkins	✓	ben-cluster	jenkins	2021/04/07 16:20 UTC	Unmanaged Manage Ignore
jenkins-jenkins	✓	ben-cluster	jenkins app.kubernetes.io/name: jenkins +1	2021/04/07 16:20 UTC	Unmanaged
jenkins2-jenkins	✓	ben-cluster	jenkins app.kubernetes.io/name: jenkins +1	2021/04/07 19:26 UTC	Unmanaged

- If you don't want to manage an app, select the drop-down list in the **Actions** column for the desired app and select **Ignore**.

For example, if you wanted to manage all apps under the "jenkins" namespace together so that they have the same snapshot and backup policies, you would manage the namespace and ignore the apps in the namespace:



Result

Apps that you chose to manage are now available from the **Managed** tab. Any ignored apps will move to the **Ignored** tab. Ideally, the Discovered tab will show zero apps, so that as new apps are installed, they are easier to find and manage.

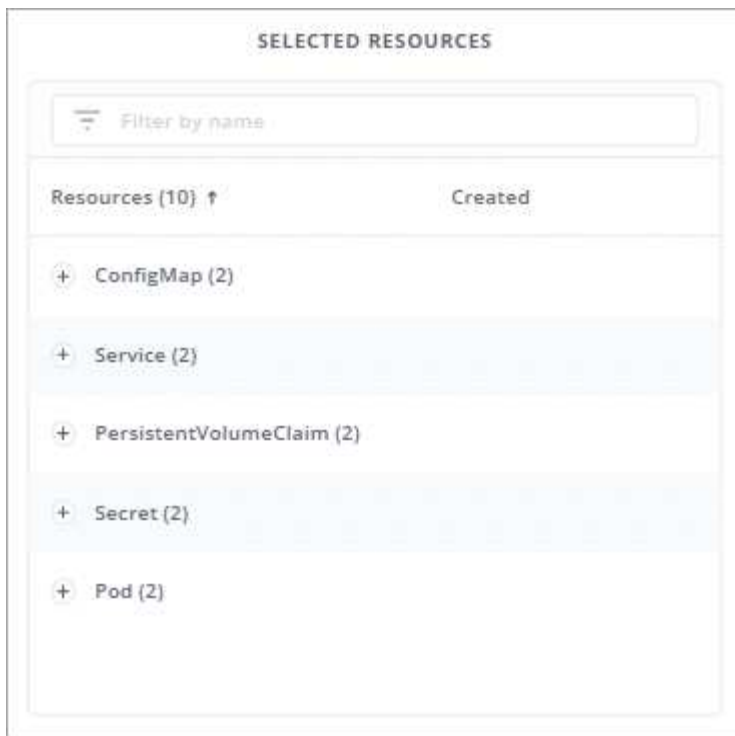
Manage apps by Kubernetes label

Astra Control includes an action at the top of the Apps page named **Define custom app**. You can use this action to manage apps that are identified with a Kubernetes label. [Learn more about defining apps by Kubernetes label.](#)

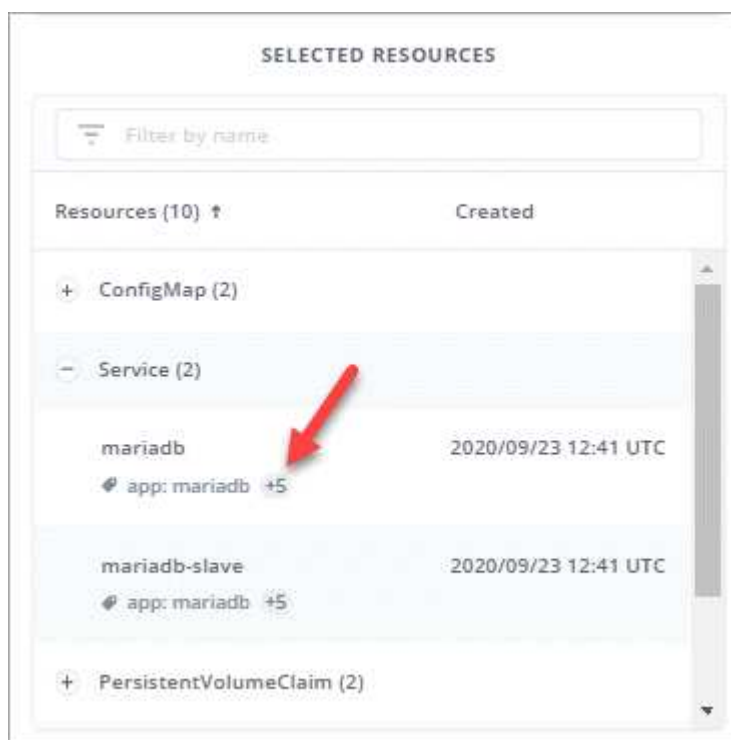
Steps

1. Select **Applications > Define custom app**.
2. In the **Define Custom Application** dialog box, provide the required information to manage the app:
 - a. **New App**: Enter the display name of the app.
 - b. **Cluster**: Select the cluster where the app resides.
 - c. **Namespace**: Select the namespace for the app.
 - d. **Label**: Enter a label or select a label from the resources below.
 - e. **Selected Resources**: View and manage the selected Kubernetes resources that you'd like to protect (pods, secrets, persistent volumes, and more).

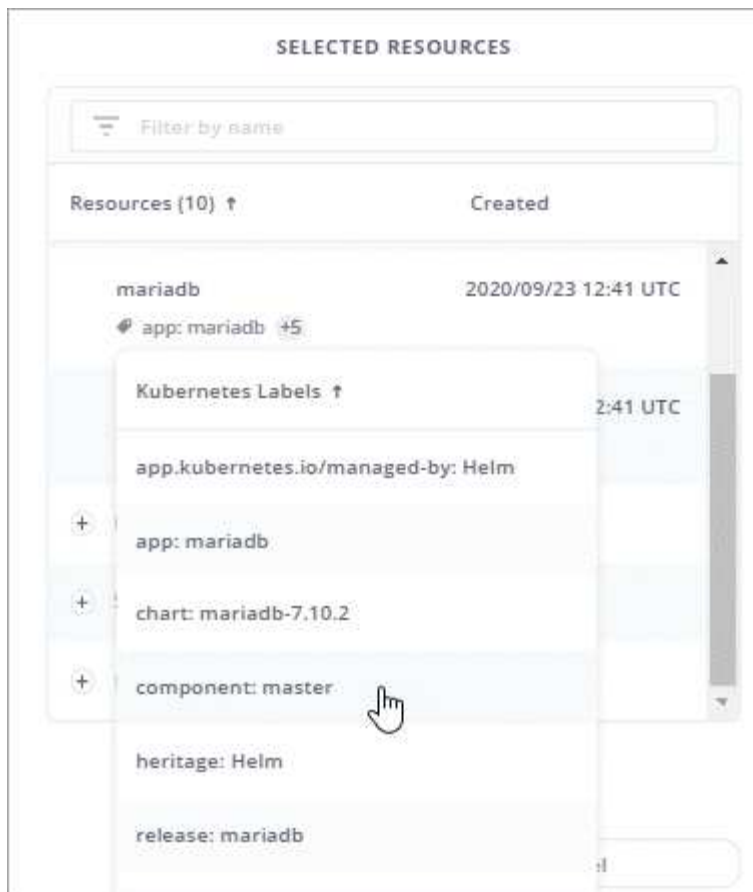
Here's an example:



- View the available labels by expanding a resource and selecting the number of labels.

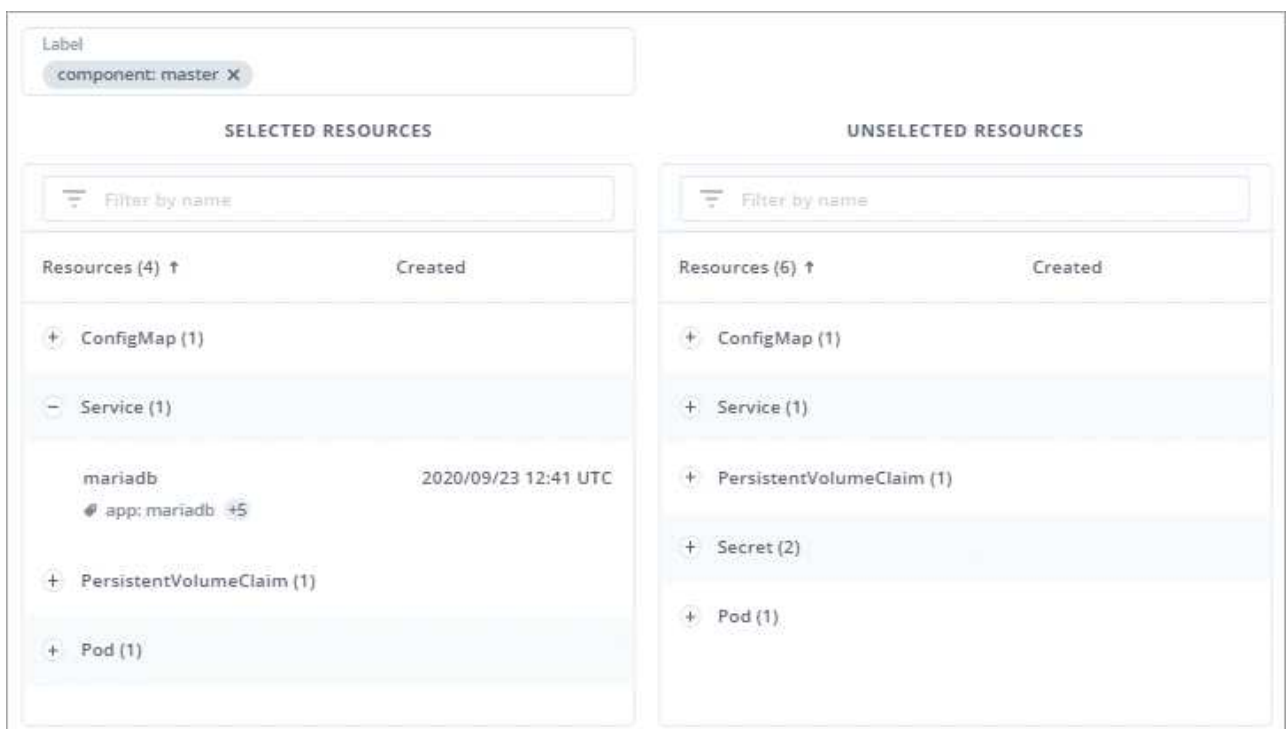


- Select one of the labels.



After you choose a label, it displays in the **Label** field. Astra Control also updates the **Unselected Resources** section to show the resources that don't match the selected label.

- f. **Unselected Resources:** Verify the app resources that you don't want to protect.



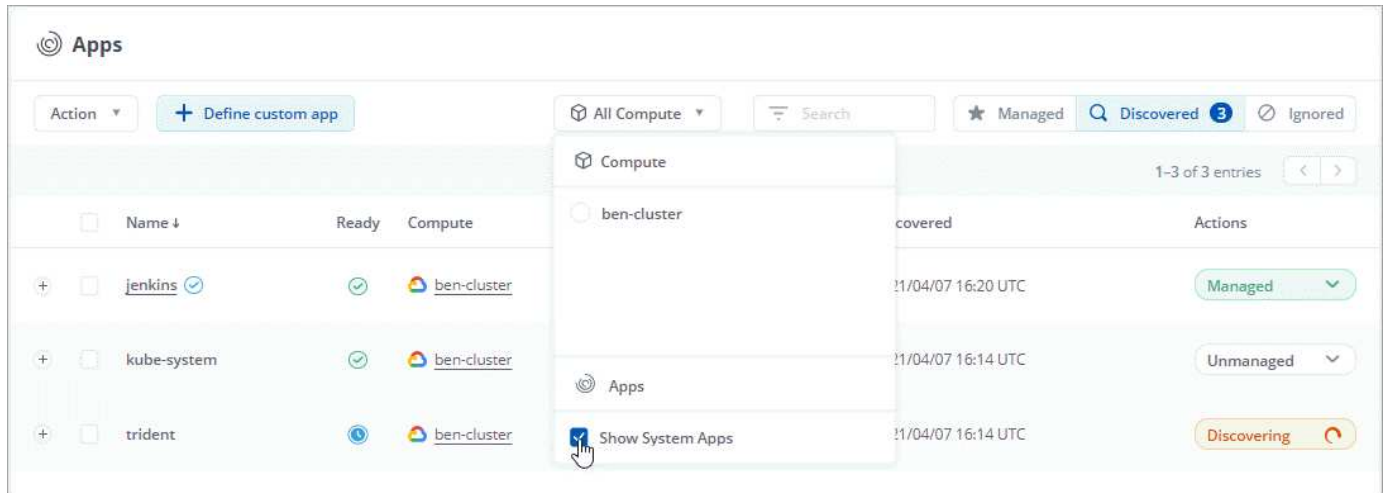
3. Select **Define Custom App**.

Result

Astra Control enables management of the app. You can now find it in the **Managed** tab.

What about system apps?

Astra Control also discovers the system apps running on a Kubernetes cluster. You can view them by filtering the Apps list.



We don't show you these system apps by default because it's rare that you'd need to back them up.

Protect apps with snapshots and backups

Protect your apps by taking snapshots and backups using an automated protection policy or on an ad-hoc basis.

Snapshots and backups

A *snapshot* is a point-in-time copy of an app that's stored on the same provisioned volume as the app. They are usually fast. Local snapshots are used to restore the application to an earlier point in time.

A *backup* is stored on object storage in the cloud. A backup can be slower to take compared to the local snapshots. But they can be accessed across regions in the cloud to enable app migrations. You can also choose a longer retention period for backups.



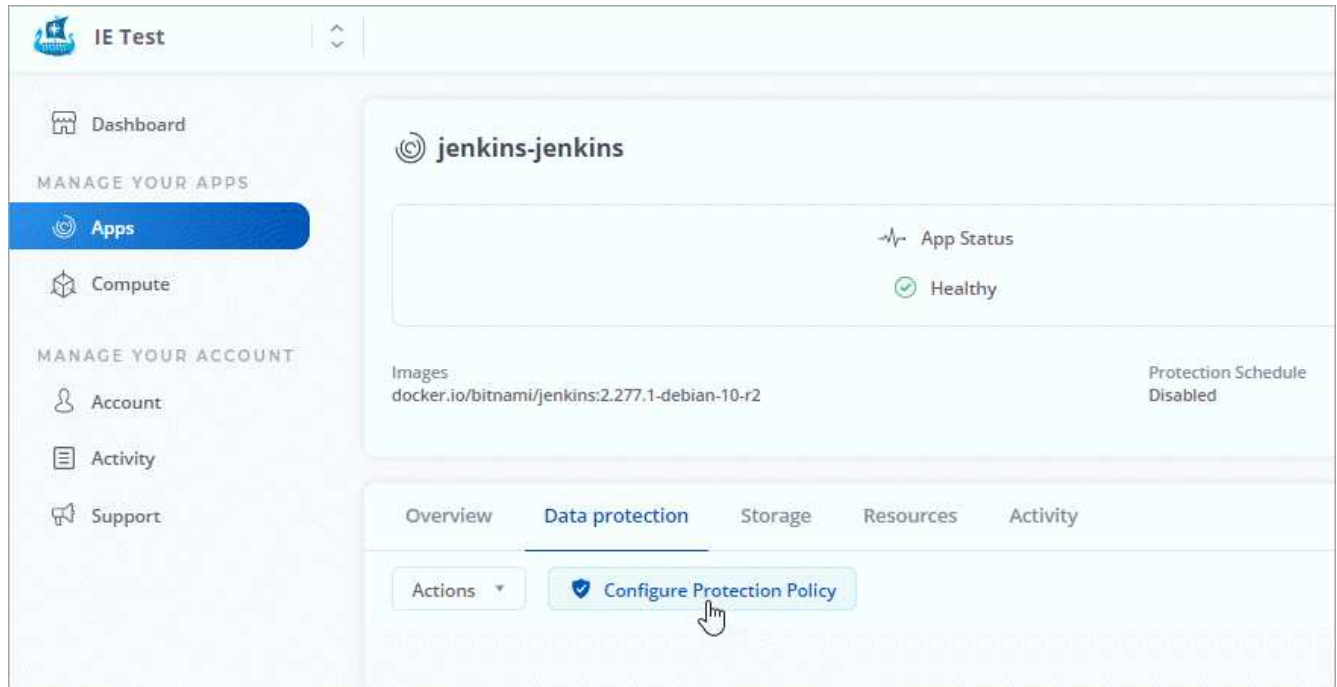
You can't be fully protected until you have a recent backup. This is important because backups are stored in an object store away from the persistent volumes. If a failure or accident wipes out the cluster and its persistent storage, then you need a backup to recover. A snapshot wouldn't enable you to recover.

Configure a protection policy

A protection policy protects an app by creating snapshots, backups, or both at a defined schedule. You can choose to create snapshots and backups hourly, daily, weekly, and monthly, and you can specify the number of copies to retain.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select **Data Protection**.
3. Select **Configure Protection Policy**.



4. Define a protection schedule by choosing the number of snapshots and backups to keep for the hourly, daily, weekly, and monthly schedules.

You can define the hourly, daily, weekly, and monthly schedules concurrently. A schedule won't turn active until you set a retention level for snapshots and backups.

When you set a retention level for backups, you can choose the bucket where you'd like to store the backups.

The following example sets four protection schedules: hourly, daily, weekly, and monthly for snapshots and backups.

Configure protection policy

STEP 1/2: DETAILS

×

PROTECTION SCHEDULE

Hourly

Every hour on the 0th minute, keep the last 4 snapshots

Daily

Daily at 05:00 (UTC), keep the last 7 snapshots

Weekly

Weekly on Mondays at 05:00 (UTC), keep the last 12 snapshots

Monthly

Every 1st of the month at 05:00 (UTC), keep the last 12 backups

Hourly
Daily
Weekly
Monthly

Day(s) of Month (optional)
Time (UTC) (optional)
Snapshots to keep
Backups to keep

1 x
05:00
0
12

BACKUP DESTINATION

Bucket:
ben-astra-bucket
Default

Cancel

Review →

OVERVIEW

Schedule and retention

Define a policy to continuously protect your application on a schedule and configure a retention count to get started.

For select stateful applications, expect I/O to pause for a short time during a backup or snapshot operation.

Read more in [Protection policies](#)

Application maria

Namespace maria

Cluster david-ie-00

5. Select **Review**.
6. Select **Configure**.

Here's a video that shows each of these steps.

► <https://docs.netapp.com/us-en/astra-control-service/media/use/video-set-protection-policy.mp4> (video)

Result

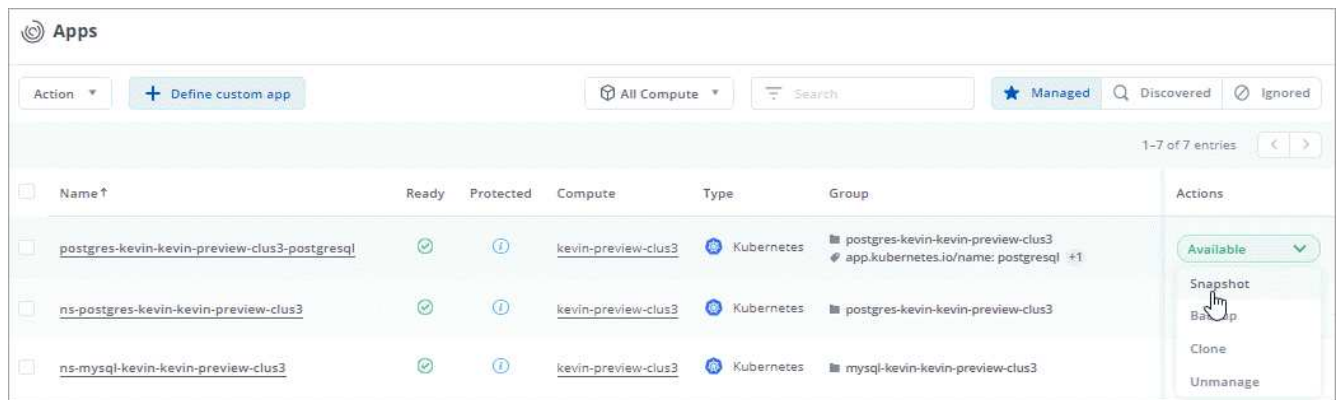
Astra Control implements the data protection policy by creating and retaining snapshots and backups using the schedule and retention policy that you defined.

Create a snapshot

You can create an on-demand snapshot at any time.

Steps

1. Select **Applications**.
2. Select the drop-down list in the **Actions** column for the desired app.
3. Select **Snapshot**.



4. Customize the name of the snapshot and then select **Review Information**.

5. Review the snapshot summary and select **Snapshot App**.

Result

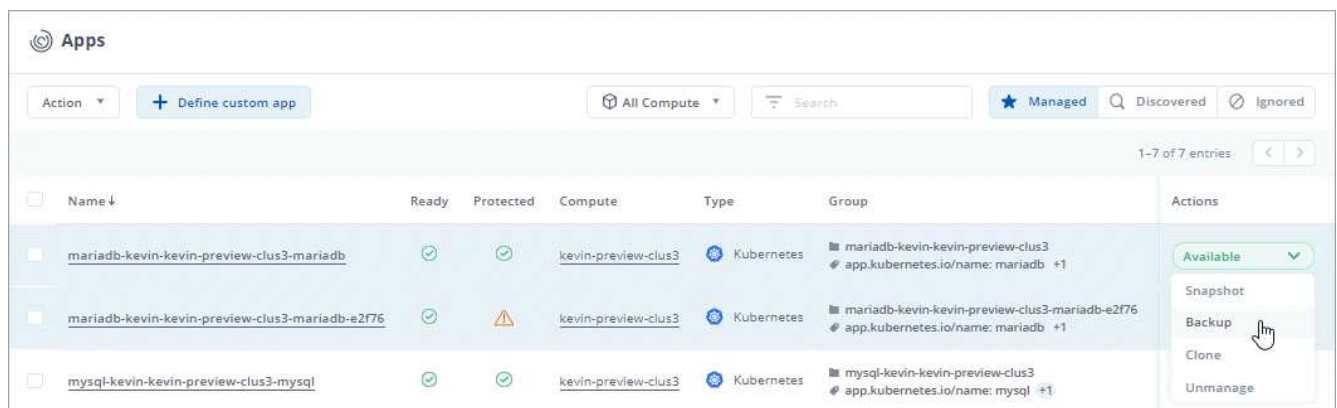
Astra Control creates a snapshot of the apps.

Create a backup

You can also back up an app at any time.

Steps

1. Select **Applications**.
2. Select the drop-down list in the **Actions** column for the desired app.
3. Select **Backup**.



4. Customize the name of the backup, choose whether to back up the app from an existing snapshot, and then select **Review Information**.

5. Review the backup summary and select **Backup App**.

Result

Astra Control creates a backup of the app.

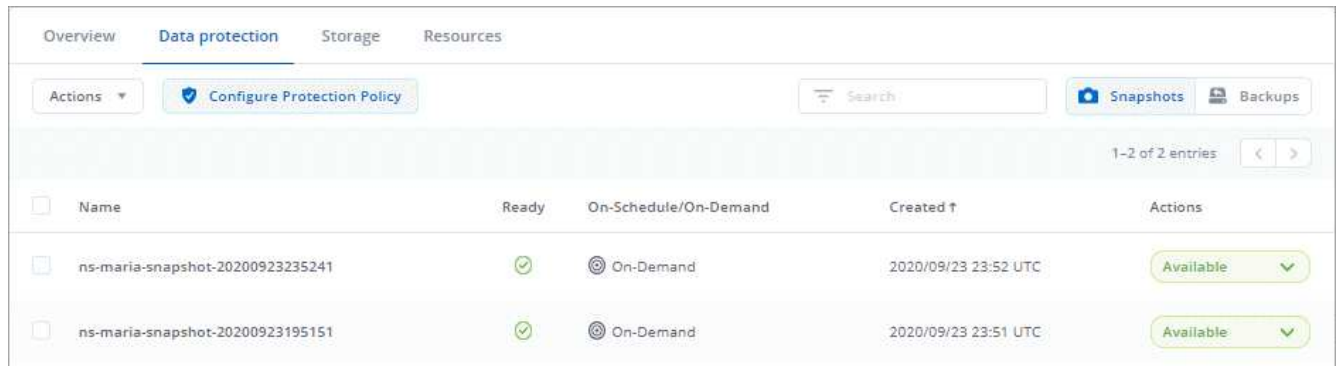
View snapshots and backups

You can view the snapshots and backups of an app from the Data Protection tab.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select **Data Protection**.

The snapshots display by default.



The screenshot shows the 'Data protection' tab in the application interface. It features a table with two entries for snapshots. The table has columns for 'Name', 'Ready', 'On-Schedule/On-Demand', 'Created', and 'Actions'. Both snapshots are in a 'Ready' state and are 'On-Demand'.

Name	Ready	On-Schedule/On-Demand	Created	Actions
ns-maria-snapshot-20200923235241	✓	On-Demand	2020/09/23 23:52 UTC	Available
ns-maria-snapshot-20200923195151	✓	On-Demand	2020/09/23 23:51 UTC	Available

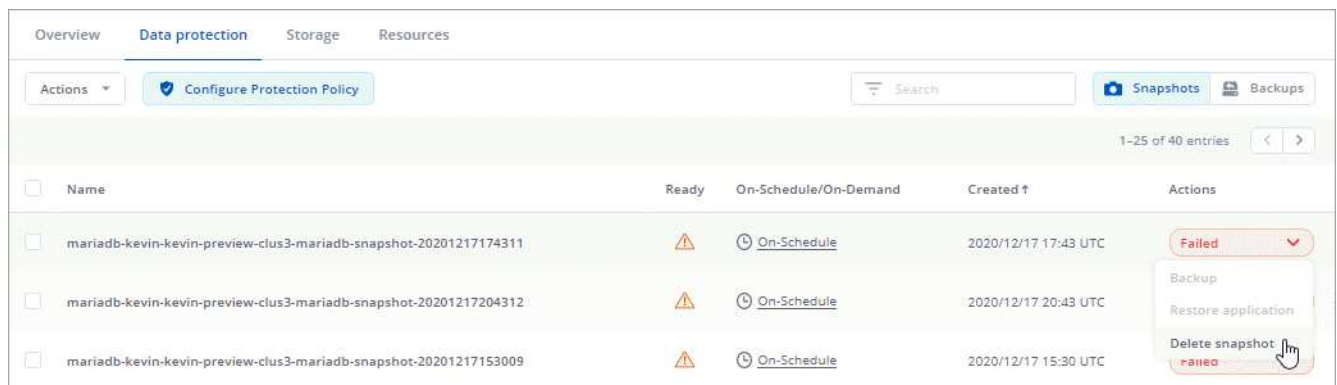
3. Select **Backups** to see the list of backups.

Delete snapshots

Delete the scheduled or on-demand snapshots that you no longer need.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select **Data Protection**.
3. Select the drop-down list in the **Actions** column for the desired snapshot.
4. Select **Delete snapshot**.



The screenshot shows the 'Data protection' tab with a table of snapshots. The 'Actions' column for the first snapshot is open, showing a dropdown menu with options: 'Backup', 'Restore application', and 'Delete snapshot'. The 'Delete snapshot' option is highlighted.

Name	Ready	On-Schedule/On-Demand	Created	Actions
mariadb-kevin-kevin-preview-clus3-mariadb-snapshot-20201217174311	⚠	On-Schedule	2020/12/17 17:43 UTC	Failed
mariadb-kevin-kevin-preview-clus3-mariadb-snapshot-20201217204312	⚠	On-Schedule	2020/12/17 20:43 UTC	
mariadb-kevin-kevin-preview-clus3-mariadb-snapshot-20201217153009	⚠	On-Schedule	2020/12/17 15:30 UTC	

5. Type the name of the snapshot to confirm deletion and then select **Yes, Delete snapshot**.

Result

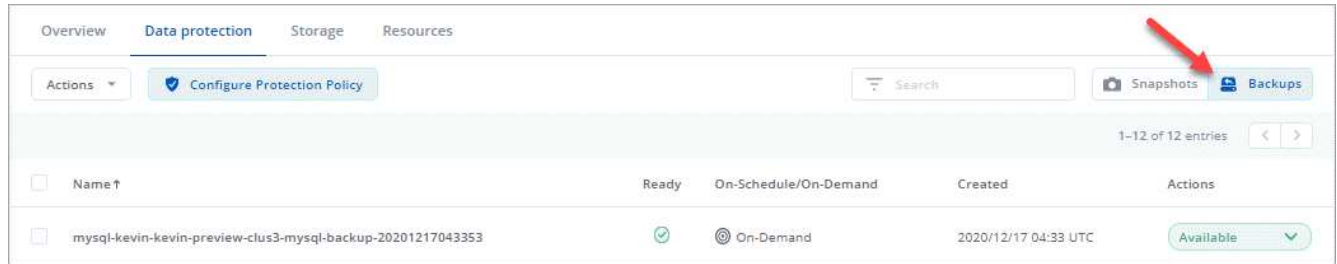
Astra Control deletes the snapshot.

Delete backups

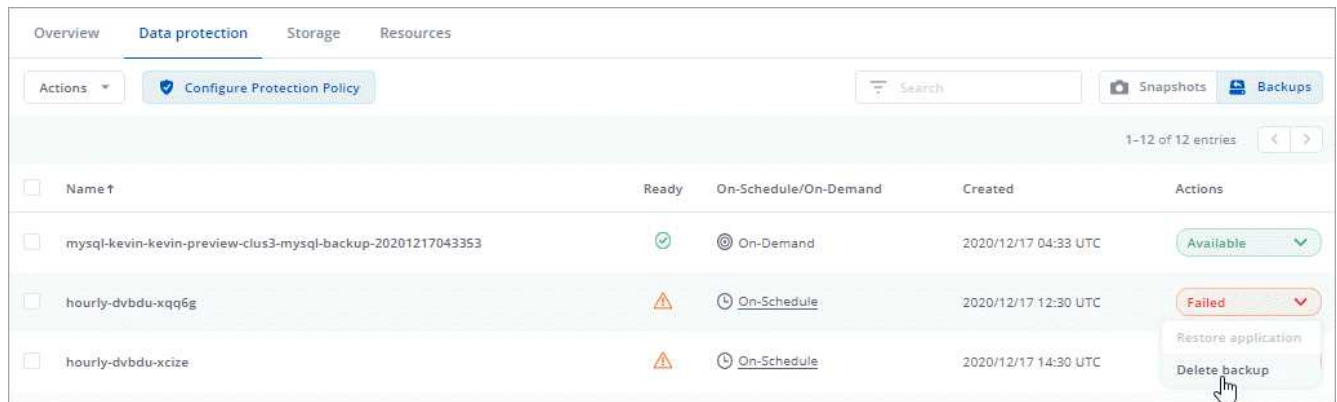
Delete the scheduled or on-demand backups that you no longer need.

1. Select **Applications** and then select the name of a managed app.

2. Select **Data Protection**.
3. Select **Backups**.



4. Select the drop-down list in the **Actions** column for the desired backup.
5. Select **Delete backup**.



6. Type the name of the backup to confirm deletion and then select **Yes, Delete backup**.

Result

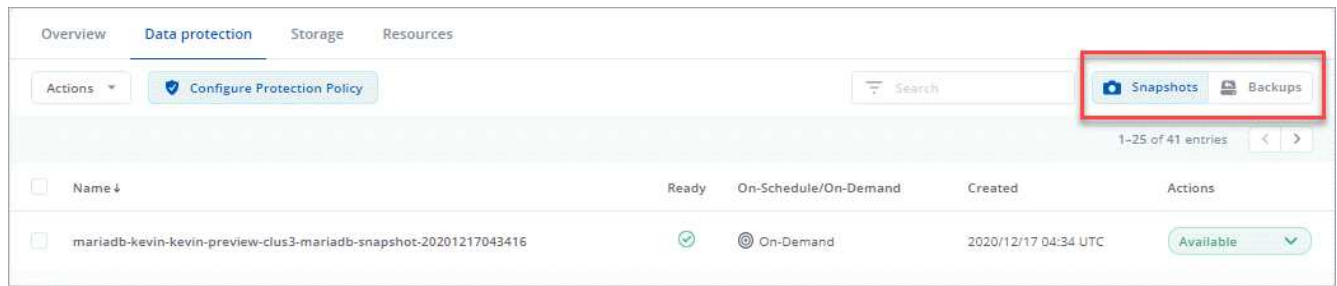
Astra Control deletes the backup.

Restore apps

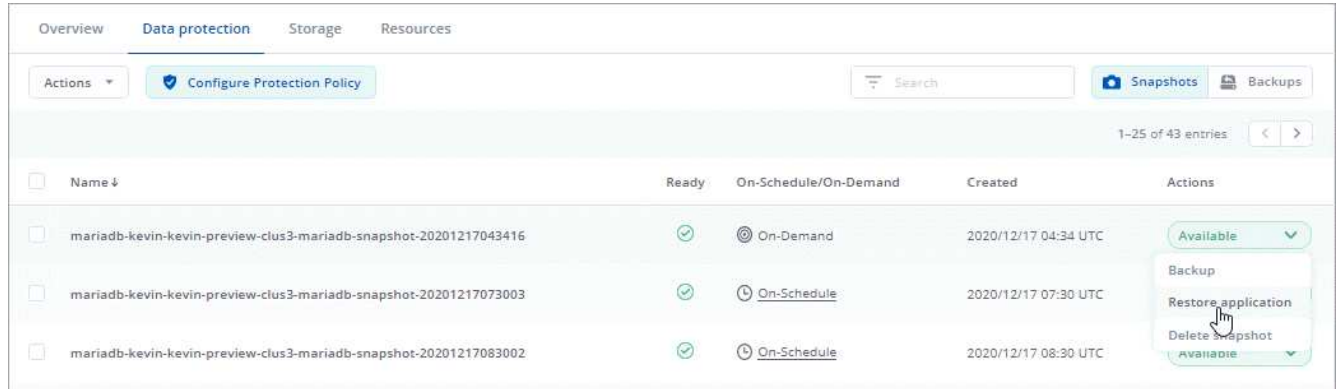
Astra Control can restore your application configuration and persistent storage from a snapshot or backup. Persistent storage backups are transferred from your object store, so restoring from an existing backup will complete the fastest.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select **Data protection**.
3. If you want to restore from a snapshot, keep **Snapshots** selected. Otherwise, select **Backups** to restore from a backup.



4. Select the drop-down list in the **Actions** column for the snapshot or backup from which you want to restore.
5. Select **Restore application**.



6. **Restore details:** Specify details for the restored app. To restore an app in-place (revert an app to an earlier version of itself), choose the same namespace and destination cluster that it is currently running in:
 - Enter a namespace for the app.
 - Choose the destination cluster for the app.
 - Click **Review**.
7. **Restore Summary:** Review details about the restore action, type "restore", and select **Restore**.

Result

Astra Control restores the app based on the information that you provided. If you restored the app in-place, the associated persistent volumes are deleted and recreated.

Clone and migrate apps

Clone an existing app to create a duplicate app on the same Kubernetes cluster or on another cluster. Cloning can help if you need to move applications and storage from one Kubernetes cluster to another. For example, you might want to move workloads through a CI/CD pipeline and across Kubernetes namespaces.

When Astra Control clones an app, it creates a clone of your application configuration and persistent storage.



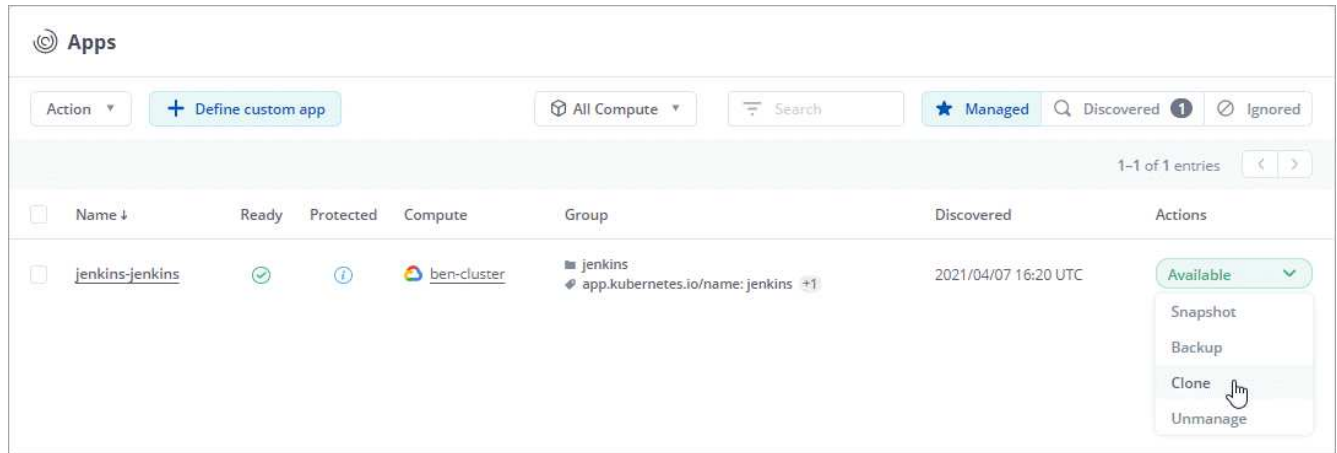
If you clone an operator-deployed instance of Jenkins CI, you need to manually restore the persistent data. This is a limitation of the app's deployment model.



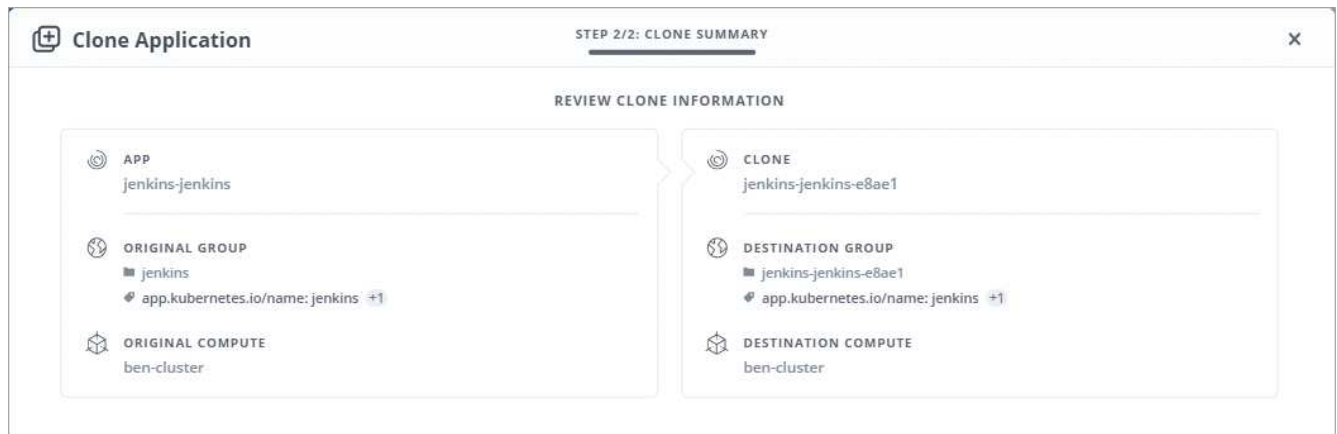
If you deploy an app with a StorageClass explicitly set and you need to clone the app, the target cluster must have the originally specified StorageClass. Cloning an application with an explicitly set StorageClass to a cluster that does not have the same StorageClass will fail.

Steps

1. Select **Applications**.
2. Select the drop-down list in the **Action** column for the desired app.
3. Select **Clone**.



4. **Clone details:** Specify details for the clone:
 - Keep the default name and namespace, or edit them.
 - Choose a destination cluster for the clone.
 - Choose whether you want to create the clone from an existing snapshot or backup. If you don't select this option, Astra Control creates the clone from the app's current state.
5. **Clone Summary:** Review the details about the clone and select **Clone App**.



Result

Astra Control clones that app based on the information that you provided.

Manage app execution hooks

An execution hook is a custom script that you can run before or after a snapshot of a managed app. For example, if you have a database app, you can use execution hooks to pause all database transactions before a snapshot, and resume transactions after the snapshot is complete. This ensures application-consistent snapshots.

Default execution hooks and regular expressions

For some apps, Astra Control comes with default execution hooks, provided by NetApp, that handle freeze and thaw operations before and after snapshots. Astra Control uses regular expressions to match an app's container image to these apps:

- MariaDB
 - Matching regular expression: `\bmariadb\b`
- MySQL
 - Matching regular expression: `\bmysql\b`
- PostgreSQL
 - Matching regular expression: `\bpostgresql\b`

If there is a match, the NetApp-provided default execution hooks for that app appear in the app's list of active execution hooks, and those hooks run automatically when snapshots of that app are taken. If one of your custom apps has a similar image name that happens to match one of the regular expressions (and you don't want to use the default execution hooks), you can either change the image name, or disable the default execution hook for that app and use a custom hook instead.

You cannot delete or modify the default execution hooks.

Important notes about custom execution hooks

Consider the following when planning execution hooks for your apps.

- Astra Control requires execution hooks to be written in the format of executable shell scripts.
- Script size is limited to 128KB.
- Astra Control uses execution hook settings and any matching criteria to determine which hooks are applicable to a snapshot.
- All execution hook failures are soft failures; other hooks and the snapshot are still attempted even if a hook fails. However, when a hook fails, a warning event is recorded in the **Activity** page event log.
- To create, edit, or delete execution hooks, you must be a user with Owner, Admin, or Member permissions.
- If an execution hook takes longer than 25 minutes to run, the hook will fail, creating an event log entry with a return code of "N/A". Any affected snapshot will time out and be marked as failed, with a resulting event log entry noting the timeout.



Since execution hooks often reduce or completely disable the functionality of the application they are running against, you should always try to minimize the time your custom execution hooks take to run.

When a snapshot is run, execution hook events take place in the following order:

1. Any applicable NetApp-provided default pre-snapshot execution hooks are run on the appropriate containers.
2. Any applicable custom pre-snapshot execution hooks are run on the appropriate containers. You can create and run as many custom pre-snapshot hooks as you need, but the order of execution of these hooks before the snapshot is neither guaranteed nor configurable.
3. The snapshot is performed.
4. Any applicable custom post-snapshot execution hooks are run on the appropriate containers. You can create and run as many custom post-snapshot hooks as you need, but the order of execution of these hooks after the snapshot is neither guaranteed nor configurable.
5. Any applicable NetApp-provided default post-snapshot execution hooks are run on the appropriate containers.



You should always test your execution hook scripts before enabling them in a production environment. You can use the 'kubectl exec' command to conveniently test the scripts. After you enable the execution hooks in a production environment, test the resulting snapshots to ensure they are consistent. You can do this by cloning the app to a temporary namespace, restoring the snapshot, and then testing the app.

View existing execution hooks

You can view existing custom or NetApp-provided default execution hooks for an app.

Steps

1. Go to **Applications** and then select the name of a managed app.
2. Select the **Execution hooks** tab.

You can view all enabled or disabled execution hooks in the resulting list. You can see a hook's status, source, and when it runs (pre- or post-snapshot). To view event logs surrounding execution hooks, go to the **Activity** page in the left-side navigation area.

Create a custom execution hook

You can create a custom execution hook for an app. See [Execution hook examples](#) for hook examples. You need to have Owner, Admin, or Member permissions to create execution hooks.



When you create a custom shell script to use as an execution hook, remember to specify the appropriate shell at the beginning of the file, unless you are running linux commands or providing the full path to an executable.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select the **Execution hooks** tab.
3. Select **Add a new hook**.
4. In the **Hook Details** area, depending on when the hook should run, choose **Pre-Snapshot** or **Post-Snapshot**.
5. Enter a unique name for the hook.
6. (Optional) Enter any arguments to pass to the hook during execution, pressing the Enter key after each

argument you enter to record each one.

7. In the **Container Images** area, if the hook should run against all container images contained within the application, enable the **Apply to all container images** check box. If instead the hook should act only on one or more specified container images, enter the container image names in the **Container image names to match** field.
8. In the **Script** area, do one of the following:
 - Upload a custom script.
 - a. Select the **Upload file** option.
 - b. Browse to a file and upload it.
 - c. Give the script a unique name.
 - d. (Optional) Enter any notes other administrators should know about the script.
 - Paste in a custom script from the clipboard.
 - a. Select the **Paste from clipboard** option.
 - b. Select the text field and paste the script text into the field.
 - c. Give the script a unique name.
 - d. (Optional) Enter any notes other administrators should know about the script.
9. Select **Add hook**.

Disable an execution hook

You can disable an execution hook if you want to temporarily prevent it from running before or after a snapshot of an app. You need to have Owner, Admin, or Member permissions to disable execution hooks.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select the **Execution hooks** tab.
3. Select the **Actions** dropdown for a hook that you wish to disable.
4. Select **Disable**.

Delete an execution hook

You can remove an execution hook entirely if you no longer need it. You need to have Owner, Admin, or Member permissions to delete execution hooks.

Steps

1. Select **Applications** and then select the name of a managed app.
2. Select the **Execution hooks** tab.
3. Select the **Actions** dropdown for a hook that you wish to delete.
4. Select **Delete**.

Execution hook examples

Use the following examples to get an idea of how to structure your execution hooks. You can use these hooks as templates, or as test scripts.

Simple success example

This is an example of a simple hook that succeeds and writes a message to standard output and standard error.

```
#!/bin/sh

# success_sample.sh
#
# A simple noop success hook script for testing purposes.
#
# args: None
#

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}

#
# main
#
```

```
# log something to stdout
info "running success_sample.sh"

# exit with 0 to indicate success
info "exit 0"
exit 0
```

Simple success example (bash version)

This is an example of a simple hook that succeeds and writes a message to standard output and standard error, written for bash.

```
#!/bin/bash

# success_sample.bash
#
# A simple noop success hook script for testing purposes.
#
# args: None

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
```

```

error() {
    msg "ERROR: $" 1>&2
}

#
# main
#

# log something to stdout
info "running success_sample.bash"

# exit with 0 to indicate success
info "exit 0"
exit 0

```

Simple success example (zsh version)

This is an example of a simple hook that succeeds and writes a message to standard output and standard error, written for Z shell.

```

#!/bin/zsh

# success_sample.zsh
#
# A simple noop success hook script for testing purposes.
#
# args: None
#

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#

```



```

info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}

#
# main
#

# log something to stdout
info "running success_sample.zsh"

# exit with 0 to indicate success
info "exit 0"
exit 0

```

Success with arguments example

The following example demonstrates how you can use args in a hook.

```

#!/bin/sh

# success_sample_args.sh
#
# A simple success hook script with args for testing purposes.
#
# args: Up to two optional args that are echoed to stdout
#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

```

```

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}

#
# main
#

# log something to stdout
info "running success_sample_args.sh"

# collect args
arg1=$1
arg2=$2

# output args and arg count to stdout
info "number of args: $#"
```

```

info "arg1 ${arg1}"
info "arg2 ${arg2}"

# exit with 0 to indicate success
info "exit 0"
exit 0

```

Pre-snapshot / post-snapshot hook example

The following example demonstrates how the same script can be used for both a pre-snapshot and a post-snapshot hook.

```
#!/bin/sh
```

```

# success_sample_pre_post.sh
#
# A simple success hook script example with an arg for testing purposes
# to demonstrate how the same script can be used for both a prehook and
posthook
#
# args: [pre|post]

# unique error codes for every error case
ebase=100
eusage=$((ebase+1))
ebadstage=$((ebase+2))
epre=$((ebase+3))
epost=$((ebase+4))

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}

```

```

#
# Would run prehook steps here
#
prehook() {
    info "Running noop prehook"
    return 0
}

#
# Would run posthook steps here
#
posthook() {
    info "Running noop posthook"
    return 0
}

#
# main
#

# check arg
stage=$1
if [ -z "${stage}" ]; then
    echo "Usage: $0 <pre|post>"
    exit ${eusage}
fi

if [ "${stage}" != "pre" ] && [ "${stage}" != "post" ]; then
    echo "Invalid arg: ${stage}"
    exit ${ebadstage}
fi

# log something to stdout
info "running success_sample_pre_post.sh"

if [ "${stage}" = "pre" ]; then
    prehook
    rc=$?
    if [ ${rc} -ne 0 ]; then
        error "Error during prehook"
    fi
fi

if [ "${stage}" = "post" ]; then
    posthook

```

```

    rc=$?
    if [ ${rc} -ne 0 ]; then
        error "Error during posthook"
    fi
fi
exit ${rc}

```

Failure example

The following example demonstrates how you can handle failures in a hook.

```

#!/bin/sh

# failure_sample_arg_exit_code.sh
#
# A simple failure hook script for testing purposes.
#
# args: [the exit code to return]
#

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#

```

```

error() {
    msg "ERROR: $" 1>&2
}

#
# main
#

# log something to stdout
info "running failure_sample_arg_exit_code.sh"

argexitcode=$1

# log to stderr
error "script failed, returning exit code ${argexitcode}"

# exit with specified exit code
exit ${argexitcode}

```

Verbose failure example

The following example demonstrates how you can handle failures in a hook, with more verbose logging.

```

#!/bin/sh

# failure_sample_verbose.sh
#
# A simple failure hook script with args for testing purposes.
#
# args: [The number of lines to output to stdout]

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#

```

```

# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}

#
# main
#

# log something to stdout
info "running failure_sample_verbose.sh"

# output arg value to stdout
linecount=$1
info "line count ${linecount}"

# write out a line to stdout based on line count arg
i=1
while [ "$i" -le ${linecount} ]; do
    info "This is line ${i} from failure_sample_verbose.sh"
    i=$(( i + 1 ))
done

error "exiting with error code 8"
exit 8

```

Failure with an exit code example

The following example demonstrates a hook failing with an exit code.

```

#!/bin/sh

# failure_sample_arg_exit_code.sh
#

```

```

# A simple failure hook script for testing purposes.
#
# args: [the exit code to return]
#

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}

#
# main
#

# log something to stdout
info "running failure_sample_arg_exit_code.sh"

argexitcode=$1

# log to stderr
error "script failed, returning exit code ${argexitcode}"

```



```
# exit with specified exit code
exit ${argexitcode}
```

Success after failure example

The following example demonstrates a hook failing the first time it is run, but succeeding after the second run.

```
#!/bin/sh

# failure_then_success_sample.sh
#
# A hook script that fails on initial run but succeeds on second run for
# testing purposes.
#
# Helpful for testing retry logic for post hooks.
#
# args: None
#

#
# Writes the given message to standard output
#
# $* - The message to write
#
msg() {
    echo "$*"
}

#
# Writes the given information message to standard output
#
# $* - The message to write
#
info() {
    msg "INFO: $*"
}

#
# Writes the given error message to standard error
#
# $* - The message to write
#
error() {
    msg "ERROR: $*" 1>&2
}
```

```
#
# main
#

# log something to stdout
info "running failure_success sample.sh"


if [ -e /tmp/hook-test.junk ] ; then
    info "File does exist. Removing /tmp/hook-test.junk"
    rm /tmp/hook-test.junk
    info "Second run so returning exit code 0"
    exit 0
else
    info "File does not exist. Creating /tmp/hook-test.junk"
    echo "test" > /tmp/hook-test.junk
    error "Failed first run, returning exit code 5"
    exit 5
fi
```

Copyright Information

Copyright © 2021 NetApp, Inc. All rights reserved. Printed in the U.S. No part of this document covered by copyright may be reproduced in any form or by any means-graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an electronic retrieval system-without prior written permission of the copyright owner.

Software derived from copyrighted NetApp material is subject to the following license and disclaimer:

THIS SOFTWARE IS PROVIDED BY NETAPP "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WHICH ARE HEREBY DISCLAIMED. IN NO EVENT SHALL NETAPP BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

NetApp reserves the right to change any products described herein at any time, and without notice. NetApp assumes no responsibility or liability arising from the use of products described herein, except as expressly agreed to in writing by NetApp. The use or purchase of this product does not convey a license under any patent rights, trademark rights, or any other intellectual property rights of NetApp.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.277-7103 (October 1988) and FAR 52-227-19 (June 1987).

Trademark Information

NETAPP, the NETAPP logo, and the marks listed at <http://www.netapp.com/TM> are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.