



PUBLISHED · OCTOBER 4, 2026

Helm for Installing and Managing Kubernetes Components

Build a practical Helm mental model, inspect and configure charts, manage release revisions, and connect Helm operations to the Kubernetes resources they create.



This is Learn post 23 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. You already know how Kubernetes stores desired state, how controllers reconcile resources, and how role-based access control (RBAC) authorizes API requests. Helm now gives you a repeatable way to install and manage a related set of those resources as one unit.

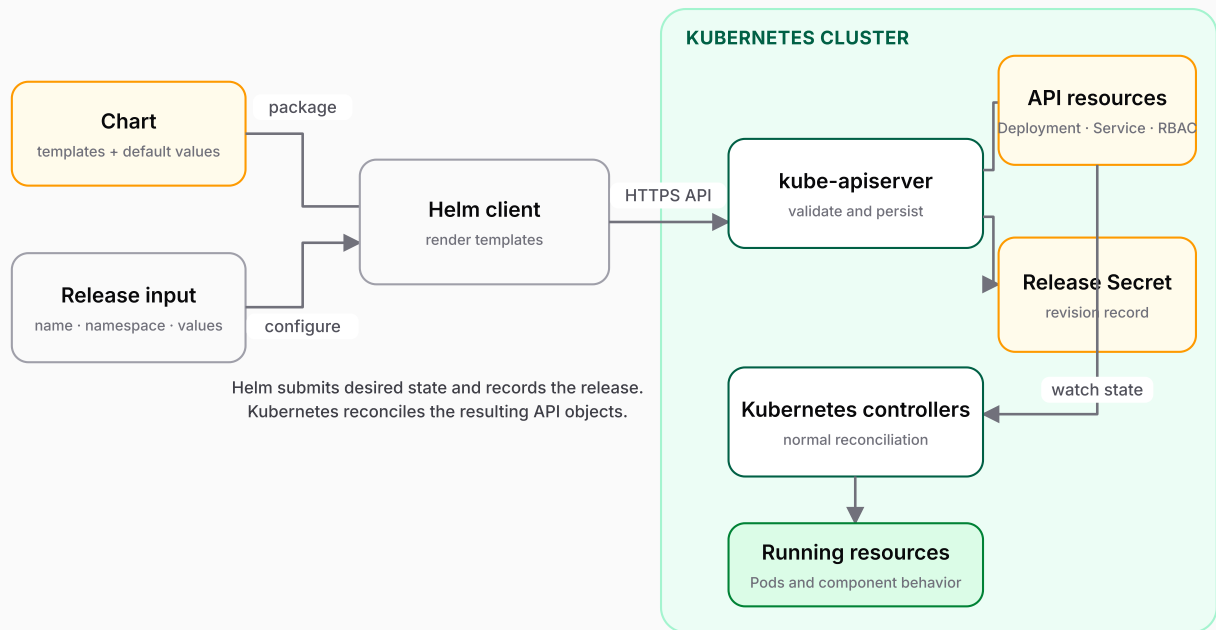
Cluster components such as metrics collectors, ingress controllers, and monitoring systems often need several Deployments, Services, ConfigMaps, ServiceAccounts, and RBAC objects. Helm packages their parameterized manifests, renders them with your configuration, and records each installation as a release.

What you'll learn

- Distinguish a chart, chart repository, release, revision, and values.
- Explain what Helm does before Kubernetes controllers begin their normal work.
- Inspect a chart, choose overrides, render manifests, and install a release safely.
- Observe a release from both the Helm and Kubernetes sides.
- Upgrade, inspect history, roll back, and uninstall while understanding the state transitions.

The mental model: render, submit, remember

A chart plus values becomes rendered Kubernetes manifests. Helm submits those manifests to the API server and remembers the result as a release revision. Kubernetes—not Helm—then reconciles the resulting objects.



Helm renders and submits; Kubernetes reconciles

Helm turns a chart and values into API objects and stores release history. Kubernetes controllers perform the ongoing reconciliation after those objects are accepted.

Helm is an API client, much like kubectl. Modern Helm does not require a server component continuously running in the cluster. After Helm creates a Deployment, the Deployment controller creates ReplicaSets and Pods exactly as it would for a Deployment created with kubectl apply.

This boundary is operationally important. Helm can report which chart configuration and revision it submitted. kubectl shows whether the resulting resources are admitted, scheduled, ready, or failing. Effective administration uses both views.

Five terms that prevent most Helm confusion

Chart: the reusable package

A chart contains metadata in `Chart.yaml`, default configuration in `values.yaml`, templates that produce Kubernetes manifests, and optionally dependencies and `CustomResourceDefinition` files. The chart is an input package, not a running object in the Kubernetes API.

Repository: where charts are published

A traditional chart repository exposes an index of chart names and versions. Helm can also pull charts from Open Container Initiative (OCI) registries. A repository is a distribution source; adding one does not install anything into Kubernetes.

Release: one installed instance

Installing a chart creates a release with a name and namespace. The same chart can be installed as different releases, each with different values. Release names are scoped to a namespace, so always keep the release name and namespace together when you inspect or change it.

Revision: one point in release history

The initial install creates revision 1. Each successful upgrade creates another revision. A rollback does not erase history; it creates a new revision whose desired configuration is based on an earlier one.

Values: the supported configuration surface

Values are inputs consumed by chart templates. The chart author chooses the available keys and their meaning; Kubernetes does not know that a field such as `replicaCount` exists. Defaults come from the chart, values files override them, and command-line `--set` options can override those for a specific operation.

Start with the target cluster, then inspect the chart

Helm uses your kubeconfig and Kubernetes credentials. Before an install, confirm the context you are about to change. Then inspect the chart's source, version, defaults, and rendered resources. A chart can create powerful ServiceAccounts and cluster-scoped RBAC, so treat it as executable deployment input rather than a harmless archive.

```
bash
```

```
kubectl config current-context  
helm version --short  
helm env
```

helm env explains where this client stores configuration and caches. It does not describe the cluster. kubectl config current-context answers the more important safety question: which API server and credentials will this operation use?

For a published chart repository, the reusable discovery pattern is add, refresh, search, then inspect. Use the URL supplied by the chart publisher and pin a reviewed chart version for repeatable installations.

```
bash
```

```
helm repo add COMPONENT_REPO https://example.com/charts  
helm repo update  
helm search repo COMPONENT_REPO/COMPONENT --versions  
helm show chart COMPONENT_REPO/COMPONENT --version CHART_VERSION  
helm show values COMPONENT_REPO/COMPONENT --version CHART_VERSION  
helm show readme COMPONENT_REPO/COMPONENT --version CHART_VERSION
```

helm repo update refreshes the client's local repository indexes; it does not upgrade installed releases. helm search repo searches those local indexes. The placeholder names above make the command pattern reusable without trusting an arbitrary third-party repository.

Worked demonstration: manage one local release

A local starter chart keeps the workflow reproducible without depending on a remote repository. The exact starter files can evolve between Helm versions, but the core structure—Chart.yaml, values.yaml, and templates—remains the same.

```
bash
```

```
helm create cka-web  
find cka-web -maxdepth 2 -type f | sort
```

```
text
```

```
cka-web/Chart.yaml  
cka-web/templates/_helpers.tpl  
cka-web/templates/deployment.yaml  
cka-web/templates/service.yaml  
cka-web/templates/serviceaccount.yaml  
cka-web/templates/tests/test-connection.yaml  
cka-web/values.yaml  
...
```

Chart.yaml identifies and versions the package. values.yaml documents the default input. Files under templates produce the API objects. Read all three layers when a chart's behavior matters; looking only at values can hide resources that are always rendered.

Choose explicit overrides

Use a values file for configuration you want to review, reuse, and keep under version control. The starter chart exposes replicaCount and service settings, so this small override is enough for the demonstration.

```
cka-values.yaml · yaml
```

```
replicaCount: 2

service:
  type: ClusterIP
  port: 80
```

Reserve `--set` for concise, intentional overrides and `--set-string` when a value must remain a string. A file is usually clearer once configuration contains several keys. Never assume a value name: discover supported keys with `helm show values` for a remote chart or read `values.yaml` for a local one.

Render before you submit

Lint catches chart-structure and template problems. Template rendering then shows the Kubernetes manifests produced by this release name, namespace, chart, and values. Neither command creates a release.

```
bash
```

```
helm lint ./cka-web -f cka-values.yaml

helm template cka-web ./cka-web \
  --namespace helm-demo \
  -f cka-values.yaml > rendered.yaml

less rendered.yaml
```

`helm template` renders locally, so it is excellent for understanding output but cannot prove the target cluster will accept it. Cluster API versions, admission policy, RBAC, quotas, and runtime conditions still matter. Also review rendered output carefully: it can contain Secret data.

When cluster access is available, server-side dry-run asks the API server to process the rendered objects without persisting them. This adds schema, authorization, and admission feedback, but it still does not prove that Pods will later become ready.

```
bash
```

```
kubectl create namespace helm-demo  
kubectl apply --dry-run=server -f rendered.yaml  
kubectl delete namespace helm-demo
```

Install revision 1

The install command names the release cka-web, targets namespace helm-demo, creates that namespace if needed, supplies the values file, and waits up to two minutes for supported readiness conditions.

```
bash
```

```
helm install cka-web ./cka-web \\  
--namespace helm-demo \\  
--create-namespace \\  
-f cka-values.yaml \\  
--wait \\  
--timeout 2m
```

```
text
```

```
NAME: cka-web  
NAMESPACE: helm-demo  
STATUS: deployed  
REVISION: 1  
...
```

The essential fields are the release name, namespace, status, and revision. --wait makes the command's success more meaningful, but only for readiness that Helm knows how to observe. The application may still require domain-specific checks beyond Kubernetes readiness.

Observe the release from both sides

Begin with Helm to answer release questions: what is installed, which revision is current, what values were supplied, and what manifests were recorded?

```
bash
```

```
helm list --namespace helm-demo
helm status cka-web --namespace helm-demo
helm get values cka-web --namespace helm-demo
helm get manifest cka-web --namespace helm-demo
helm history cka-web --namespace helm-demo
```

helm get values shows the user-supplied configuration by default; helm show values shows chart defaults. helm get manifest shows the rendered manifests recorded for the current release. These answer different questions and should not be treated as interchangeable.

Then use kubectl to inspect the actual API objects and controller state. The Helm release is not a substitute for Deployments, ReplicaSets, Pods, Services, events, or logs.

```
bash
```

```
kubectl get all --namespace helm-demo
kubectl get deployment --namespace helm-demo \
  -l app.kubernetes.io/instance=cka-web
kubectl describe deployment --namespace helm-demo \
  -l app.kubernetes.io/instance=cka-web
```

Well-formed charts commonly apply the app.kubernetes.io/instance label with the release name, making the resources easier to select. Labels are chart output rather than magic added to every object by Kubernetes, so verify them instead of assuming they exist.

By default, Helm stores release information as Kubernetes Secrets in the release namespace. That is why RBAC around Secrets affects who can inspect Helm state, and why sensitive values should not be placed casually in a chart's configuration.

bash

```
kubectl get secrets --namespace helm-demo \
  -l owner=helm \
  -o custom-columns='NAME:.metadata.name,STATUS:.metadata.labels.status,REVISION:.metadata.labels.version'
```

text

NAME	STATUS	REVISION
sh.helm.release.v1.cka-web.v1	deployed	1

The Secret is Helm's release record, not the Deployment's desired state. Deleting or editing release records directly can break Helm's view of history without correctly managing the workload. Use Helm commands for release operations.

Upgrade by declaring the next chart and values

An upgrade renders the specified chart again, compares the resulting objects with the live release, submits required changes, and records a new revision. For a remote chart, pin the chart version; for this local chart, the directory is the chart source.

cka-values.yaml · yaml

```
replicaCount: 3

service:
  type: ClusterIP
  port: 80
```

bash

```
helm upgrade cka-web ./cka-web \
--namespace helm-demo \
-f cka-values.yaml \
--wait \
--timeout 2m

helm history cka-web --namespace helm-demo
kubectl get deployment,pods --namespace helm-demo \
-l app.kubernetes.io/instance=cka-web
```

text

REVISION	STATUS	DESCRIPTION
1	superseded	Install complete
2	deployed	Upgrade complete

The exact timestamps and object names vary, but the transition is stable: revision 2 becomes deployed and revision 1 remains in history as superseded. Kubernetes then performs the normal Deployment scaling or rollout implied by the rendered manifest.

Values can become subtle during upgrades because chart defaults may change and several value sources can be merged. Make the values file and chart version explicit, inspect the result, and use `helm get values` after the operation. Reach for `--reuse-values` or `--reset-values` only when you deliberately want their documented merge behavior.

Automation commonly uses `helm upgrade --install` so the same command installs a missing release or upgrades an existing one. This is convenient, but it does not make an unsafe chart or unreviewed value change safe.

bash

```
helm upgrade --install cka-web ./cka-web \
--namespace helm-demo \
--create-namespace \
-f cka-values.yaml \
--wait \
--timeout 2m
```

Rollback creates forward history from an older state

When a previous release revision is the desired recovery point, inspect history and roll back to that exact revision. Here revision 1 used two replicas.

bash

```
helm history cka-web --namespace helm-demo

helm rollback cka-web 1 \
  --namespace helm-demo \
  --wait \
  --timeout 2m

helm history cka-web --namespace helm-demo
kubectl get deployment --namespace helm-demo \
-l app.kubernetes.io/instance=cka-web
```

text

REVISION	STATUS	DESCRIPTION
1	superseded	Install complete
2	superseded	Upgrade complete
3	deployed	Rollback to 1

Rollback to revision 1 creates revision 3. History is append-only from the administrator's perspective: the current state moved forward to a new revision based on old configuration. The API objects still change through their normal Kubernetes update paths.

Rollback can restore recorded chart configuration, but it cannot reverse every external effect. Data migrations, external systems, persistent data, and chart hooks may need component-specific recovery procedures. Read the chart's documentation before treating rollback as a universal undo button.

Uninstall removes a release, not its namespace

Uninstall asks Helm to remove the resources associated with the release and, by default, removes its release history. The namespace created with `--create-namespace` remains because it is a broader administrative boundary that may contain other objects.

```
bash
```

```
helm uninstall cka-web --namespace helm-demo
```

```
helm list --namespace helm-demo
```

```
kubectl get all --namespace helm-demo
```

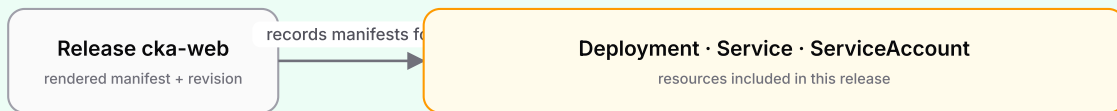
```
kubectl get namespace helm-demo
```

Some resources may intentionally remain because of chart retention policy or Kubernetes lifecycle behavior. Persistent storage and CustomResourceDefinitions require particular care; storage is covered in posts 33–35, while CRDs and Operators are taught in post 26.

Helm ownership and Kubernetes ownership are different

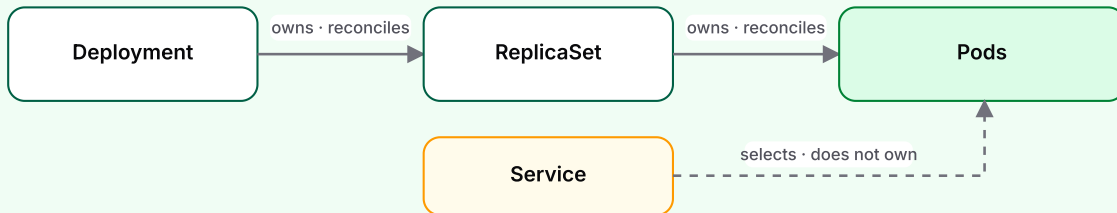
Helm manages a release from rendered manifests and release metadata. Kubernetes controllers manage runtime state from API objects. A Pod created by a chart is usually owned by a ReplicaSet, not by Helm; the ReplicaSet may be owned by a Deployment. Helm's relationship is release management, not the ownerReferences chain used by Kubernetes garbage collection.

HELM RELEASE TRACKING



This release relationship is Helm metadata—not a Kubernetes ownerReference.

KUBERNETES RELATIONSHIPS



Helm release tracking is separate from Kubernetes ownership

Helm groups resources in release metadata. Kubernetes ownerReferences connect dependent objects in controller chains; selection, such as a Service selecting Pods, is a different relationship again.

A manual `kubectl` edit can make a live object drift from the configuration recorded by Helm. A later upgrade may replace that change with newly rendered chart output. Prefer changing the release's values or chart source, then upgrading, so the intended state remains reproducible.

Post 24 introduces Kustomize, which transforms plain Kubernetes manifests without creating Helm release history. For now, remember the distinction: Helm starts with a chart and values and manages named release revisions; Kustomize starts with resources and overlays. Detailed comparison belongs to the next lesson.

Read failures in the right layer

A Helm operation crosses several boundaries. First templates must render. Then authentication, RBAC, API validation, admission, and quotas determine whether objects can be stored. Finally Kubernetes controllers, the scheduler, kubelets, networking, and storage determine whether the component becomes healthy.

text

Render error?	helm lint / helm template
API rejection?	Helm error + API message
Objects accepted?	helm status / helm get manifest
Workload unhealthy?	kubectl get / describe / logs / events

Because each layer produces different evidence, do not stop at a release status. A failed install may be an RBAC denial; a deployed release may contain an unready application if you did not wait; a timed-out wait may point to a Pending Pod whose real explanation is in scheduling events.

Use `--wait` and an intentional `--timeout` when readiness matters. For critical automation, review the Helm version's failure-handling flags and chart-specific behavior rather than assuming every failed operation automatically returns the cluster to its previous state.

Administrator checklist

- Confirm the kubeconfig context and release namespace before changing anything.
- Trust the chart source deliberately, inspect its templates and RBAC, and pin a reviewed chart version.
- Keep reusable configuration in a values file and render the result before installation.
- Use Helm commands for release state and kubectl commands for Kubernetes resource state.
- After an upgrade or rollback, verify both the active revision and the resulting workloads.
- Protect release Secrets and avoid placing sensitive material directly in ordinary values files.

Where this fits in the series

Earlier lessons supplied the object, controller, kubeconfig, cluster, and RBAC models that explain Helm's output. Post 24 next teaches Kustomize for declarative configuration without release packaging. Extension interfaces, CRDs, Operators, networking, and storage follow in their own lessons. Guided repair of Helm and related administration problems returns in Practice post 47.

Official documentation

[Introduction to Helm](#) defines charts, repositories, releases, values, and Helm's client-side architecture.

[Helm command reference](#) documents install, upgrade, rollback, list, status, get, history, template, and repository commands for the current Helm release.

[Charts](#) explains chart structure, Chart.yaml, templates, values, dependencies, and CRD directories.

[Values files](#) explains how chart defaults, user-supplied files, and command-line values are combined.

[Chart Repository Guide](#) describes repository indexes and how Helm caches repository metadata for search.