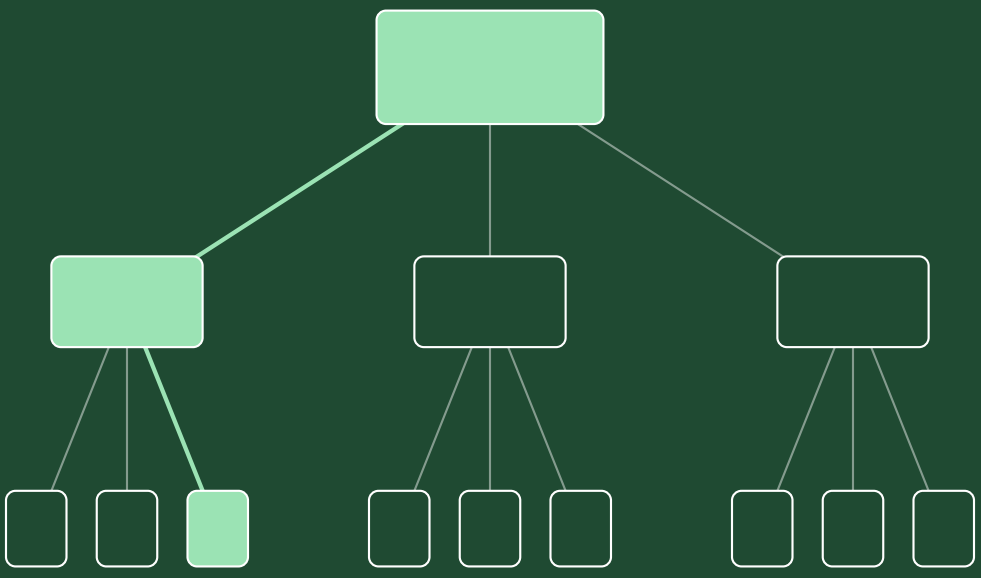
nyrvex.com

Helm Charts

Hands-On

Package, ship, and run
Kubernetes applications



Interview-ready with minimal text

Helm Charts

Hands-On

Package, ship, and run Kubernetes applications

Interview-ready with minimal text

Thomas Zachmann

Helm Charts Hands-On. Package, ship, and run Kubernetes applications.
As of November 2026. © 2026 Thomas Zachmann. License: CC BY-ND 4.0.

Written for Helm 4 (commands as of Helm 4.3). The commands, charts, and sample output in this book have not been run on a cluster. Output is illustrative, and details may vary by version. Try everything on a test cluster first, never directly in production. Domains such as `example.com` are placeholders.

Typefaces: Charter, Fira Sans, JetBrains Mono.

Contents

How to Read This Book 6

Getting Started **9**
1 What Is Helm? 10
2 Chart, Release, Repository 12
3 The Lab 14
4 The First Release 16

Building Charts **19**
5 Chart Structure 20
6 Chart.yaml 22
7 values.yaml and Precedence 24
8 The Built-In Objects 27

Thinking in Templates **29**
9 Template Fundamentals 30
10 Whitespace 32
11 if, with, range 34
12 Variables and Scope 36
13 define, include, _helpers.tpl 38
14 Sprig: The Key Functions 40
15 tpl, required, lookup 42
16 Reading Files 44

Advanced Charts **47**
17 values.schema.json 48
18 Dependencies and Subcharts 50
19 Library Charts 53
20 Hooks 55
21 Chart Tests 57
22 CRDs 59

Operations **61**
23 install, upgrade, rollback 62
24 Where Helm Stores Its Releases 65
25 3-Way Merge and Server-Side Apply 67
26 What Helm 4 Changes 69
27 OCI Registries 71
28 Signing and Provenance 73
29 Secrets 75

Quality and Delivery **79**
30 Lint and Render 80
31 helm-unittest 82
32 CI Pipeline 84
33 GitOps: Argo CD and Flux 86
34 Helm or Kustomize? 89

Troubleshooting **91**
35 The Method 92
36 pending-upgrade 94
37 Immutable Fields 97
38 Ownership and Conflicts 99

The Interview **103**

39 The Questions 104
40 Exercises 108
41 Cheat Sheet 111
Index 114

How to Read This Book

You have an interview coming up and you need to know Helm. Not someday. Soon. This book is built for exactly that: little text, lots of small examples, and for every topic the question you will actually get asked.

The rules

- **Type everything yourself.** Don't copy anything. Your fingers learn too.
- **One command at a time.** You type, you watch what happens.
- **Rendering is free.** `helm template` needs no cluster, so you can see every template example right away.
- **Read the result.** In the end, all that counts is the YAML Kubernetes receives.

> **REMEMBER**
>
> The commands, charts, and output in this book have not been run on a cluster. The output shown is shortened sample output. Your text and numbers will differ.

The boxes

> **PICTURE THIS**
>
> *An everyday comparison. If you remember it, you can work out the topic from there.*

```
helm template web ./web
```

Memory aid: A command to type, with a memory aid below it.

> **PITFALL**
>
> The mistake nearly everyone makes. And the one interviewers love to ask about.

> **IN THE INTERVIEW**
>
> **A typical question?**
>
> An answer in two or three sentences. It can be that short.

Your study plan

You have ...	Then read ...
1 evening	Chapters 1, 2, 5, 7, 11, 13, 23, 25, then chapter 39 (The Questions) and 41 (Cheat Sheet).
2 days	Day 1: parts I to III. Day 2: chapters 18, 20, 22 to 26, 36 to 38, then chapter 39.
1 week	Day 1: parts I and II. Then one part per day. Day 6: the exercises from chapter 40. Day 7: review the questions.

PART I

Getting Started

What Helm is, how it thinks, and how to install your first release.

1

What Is Helm?

The package manager for Kubernetes.

An application in Kubernetes consists of many YAML files: Deployment, Service, Ingress, ConfigMap, and more. Helm bundles them into one package. Its core idea fits on one line:

Manifests = function(templates, values)

You write the templates once. For each environment, you change only the values.

The problem

Without Helm, you have three copies of your manifests for dev, staging, and prod. They differ in five lines, and over time in many more. On top of that, Kubernetes doesn't know which objects belong together or what was installed yesterday.

Helm solves both:

- **Packaging:** Templates and default values become a chart.
- **Configuring:** Each environment adds its own values.
- **Lifecycle:** Install, upgrade, roll back, and remove as one unit.

> **PICTURE THIS**
>
> *Helm is like `apt` for Kubernetes. A chart is the package, the cluster is the machine, and `helm upgrade` is the update. Add a mail merge to that: The letter stays the same, only the recipient data changes.*

How Helm works

```
Chart + values  -->  Template Engine  -->  Manifests
                                             |
Kubernetes API  <--  Helm sends them  <-------+
```

```
                                                  |
Secret in the cluster  <--  Helm records the release
```

Helm runs as a program on your machine or in CI. There is no server in the cluster. Helm talks directly to the Kubernetes API and uses your `kubeconfig` along with your permissions.

> **REMEMBER**
>
> Helm 2 had a server in the cluster, Tiller. It was a security problem and has been gone since Helm 3. Interviewers like to ask about this.

The versions

Version	Status in November 2026
Helm 4	Current, since November 2025. This book is based on it.
Helm 3	Security updates only, until February 10, 2027.
Helm 2	Discontinued long ago.

IN THE INTERVIEW

What is Helm, and what problem does it solve?

The package manager for Kubernetes. It turns manifests into reusable templates, separates configuration into values, and manages an application as one unit, with history and rollback.

IN THE INTERVIEW

Why was Tiller removed?

Tiller ran in the cluster with broad permissions and so bypassed the permissions of individual users. Since Helm 3, the client works directly with the API and the user's RBAC permissions.

KEY TAKEAWAYS

- Manifests = function(templates, values).
- Helm runs client-side, with no server in the cluster.
- Helm 4 is current.

2

Chart, Release, Repository

Three words, and you can follow any Helm conversation.

Term	Meaning	Comparison
Chart	The package: templates, default values, metadata.	Recipe
Release	An installed instance of a chart in the cluster.	Cooked dish
Repository	A place where charts are kept.	Cookbook shelf
Revision	One version state of a release.	Number in the logbook

A chart can be installed many times. Each installation is a separate release with its own name. That's how `shop-a` and `shop-b` run from the same chart.

Every `upgrade` and every `rollback` creates a new revision. So a rollback doesn't go "back." It writes the old state into the logbook under a new number.

```
Revision 1  install    v1.0
Revision 2  upgrade    v1.1
Revision 3  rollback   v1.0 (content of revision 1)
```

Two versions in Chart.yaml

- `version` is the version of the **chart**.
- `appVersion` is the version of the **application** inside it.

The two are independent. If you change only a template, `version` goes up while `appVersion` stays the same.

Where charts come from

- **OCI registry:** like container images. Today's standard, see chapter 27.
- **HTTP repository:** a web server with an `index.yaml`.
- **Local:** a folder or a `.tgz` file.
- **Artifact Hub:** a search engine for public charts.

IN THE INTERVIEW

What is the difference between a chart and a release?

The chart is the package, the release is its running instance in the cluster. One chart can be installed as many releases, even in the same namespace, as long as the names differ.

IN THE INTERVIEW

`version` or `appVersion`?

`version` is the chart's SemVer version and identifies the package. `appVersion` describes the included application and is purely informational, though templates often use it as the default image tag.

KEY TAKEAWAYS

- The chart is the recipe, the release is the dish.
- Every change produces a new revision.
- `version` belongs to the chart, `appVersion` to the application.

3

The Lab

A local cluster and the Helm binary. That's all you need.

A laptop is enough for this book. You need Docker, `kubectl`, `kind`, and `helm`.

```
kind create cluster --name helm-lab
```

Memory aid: **kind** stands for **K**ubernetes **in** **D**ocker: a cluster in containers.

```
kubectl get nodes
```

Memory aid: Is your cluster answering?

```
helm version
```

Memory aid: You're asking Helm for its ID. It should start with a 4.

Who does Helm talk to?

Helm uses the same context as `kubectl`.

```
kubectl config current-context
```

Memory aid: Am I in the right cluster? Ask yourself this before every `upgrade`.

```
helm env
```

Memory aid: Shows paths for cache, configuration, and data.

> **PITFALL**
>
> Helm uses the current context without warning. If you were just in production, that's where you install. In scripts, always set `--kube-context` and `--namespace` explicitly.

A chart to play with

```
helm create web
```

Memory aid: **create** generates a ready-made sample chart in the `web` folder.

```
helm template web ./web
```

Memory aid: Render only, send nothing to the cluster. Your most important command while learning.

KEY TAKEAWAYS

- `kind`, `kubectl`, `helm`.
- Check the context before every change.
- `helm template` shows the result without a cluster.

4

The First Release

Install, inspect, change, roll back, remove.

Install

```
helm install web ./web --namespace demo --create-namespace
```

Memory aid: Read it as a sentence: Install the chart from `./web` as release `web` in the namespace `demo`.

```
NAME: web
NAMESPACE: demo
STATUS: deployed
REVISION: 1
```

Inspect

```
helm list -n demo
```

Memory aid: Which releases exist here? `-A` shows all namespaces.

```
helm status web -n demo
```

Memory aid: How is the release doing, and what do the notes say?

```
helm get values web -n demo
```

Memory aid: Which values did **I** set?

```
helm get manifest web -n demo
```

Memory aid: What did Helm actually send to the cluster?

Change

```
helm upgrade web ./web -n demo --set replicaCount=3
```

Memory aid: New values, new revision. Helm changes only what differs.

```
helm history web -n demo
```

Memory aid: The logbook of all revisions.

Roll back

```
helm rollback web 1 -n demo
```

Memory aid: Back to the content of revision 1. Without a number, it goes to the previous one.

Remove

```
helm uninstall web -n demo
```

Memory aid: Removes the release's objects and its history.

> **PITFALL**
>
> `helm uninstall` also deletes the history. After that, there's no rollback. With `--keep-history`, the logbook survives. Also, PVCs from a StatefulSet's `volumeClaimTemplates` stay behind, because they don't belong to the release.

Using third-party charts

```
helm show values oci://registry.example.com/charts/web
```

Memory aid: First read what can be configured. `helm show all` gives you everything.

```
helm pull oci://registry.example.com/charts/web --version
1.2.0 --untar
```

Memory aid: Get the chart onto your disk and look inside before you trust it.

IN THE INTERVIEW

What happens during `helm install`?

Helm loads the chart, merges values, renders the templates, and validates them. Then it sends the manifests to the API and stores the release as a Secret in the namespace.

KEY TAKEAWAYS

- `install`, `list`, `status`, `upgrade`, `history`, `rollback`, `uninstall`.
- `helm get` shows what's stored in the release.
- Read third-party charts before you install them.

PART II

Building Charts

Folders, metadata, values, and the built-in objects.

5

Chart Structure

A chart is just a folder with fixed names.

```
web/
├── Chart.yaml          Name, version, dependencies
├── Chart.lock          Pinned dependencies
├── values.yaml         Default values
├── values.schema.json  Validation rules for values (optional)
├── charts/             Dependencies
├── crds/               CRDs (optional)
├── templates/
│   ├── NOTES.txt       Text shown after installation
│   ├── _helpers.tpl    Your own building blocks
│   ├── deployment.yaml
│   ├── service.yaml
│   └── tests/          Chart tests
├── .helmignore         What stays out of the package
└── README.md
```

Rule	Meaning
Everything in `templates/`	is rendered and installed.
Files starting with `_`	produce no manifest. They contain only building blocks.
`NOTES.txt`	is rendered and displayed after installation.
`templates/tests/`	contains Pods that `helm test` starts.

A template should contain exactly one Kubernetes object. The file name states the kind, for example `deployment.yaml`.

> **PICTURE THIS**
>
> *A chart is like a suitcase with fixed compartments: metadata in the outer pocket, templates in the main compartment, small items in the pouch with the underscore.*

Packaging

```
helm package ./web
```

Memory aid: Packs the folder into `web-1.0.0.tgz`. The name comes from `Chart.yaml`.

```
helm show chart web-1.0.0.tgz
```

Memory aid: Read the metadata from the package.

IN THE INTERVIEW

What is `_helpers.tpl` for?

It holds named templates the chart reuses, for example for names and labels. The underscore tells Helm: This file produces no manifest.

KEY TAKEAWAYS

- `Chart.yaml`, `values.yaml`, `templates/`.
- The underscore means: building blocks only.
- One object per file.

6

Chart.yaml

The package's ID card.

Chart.yaml

```yaml
apiVersion: v2
name: web
description: A small web service
type: application
version: 1.4.2
appVersion: "2.7.0"
kubeVersion: ">=1.30.0-0"
maintainers:
  - name: Platform Team
    email: platform@example.com
```

Field	Meaning
`apiVersion`	The chart format. `v2` is the standard.
`name`	The name of the chart.
`type`	`application` or `library`, see chapter 19.
`version`	Chart version, must be SemVer.
`appVersion`	Version of the application. Best kept in quotes.
`kubeVersion`	Which Kubernetes versions the chart tolerates.
`dependencies`	Other charts, see chapter 18.

SemVer for charts

- **Major:** breaking. Users have to do something, for example rename values.
- **Minor:** new feature, backward compatible.
- **Patch:** bug fix.

> **PITFALL**
>
> YAML reads `appVersion: 2.10` without quotes as a number, so `2.1`. Always write versions as strings.

> **REMEMBER**
>
> There are also experimental charts with `apiVersion: v3`. They're hidden behind `HELM_EXPERIMENTAL_CHART_V3=1` and may still change. For real work, `v2` remains correct, and existing charts keep working under Helm 4.

IN THE INTERVIEW

When do you bump which part of the chart version?

At least patch for every chart change. Minor for new options, major when users have to adapt something, such as renamed values or removed options.

KEY TAKEAWAYS

- `apiVersion: v2` is the standard.
- `version` is SemVer.
- Versions in quotes.

7

values.yaml and Precedence

Who wins when several sources set the same value?

`values.yaml` is the default configuration and the most important documentation of your chart.

`values.yaml`

```yaml
# Number of Pods
replicaCount: 2

image:
  repository: registry.example.com/web
  # Defaults to appVersion from Chart.yaml
  tag: ""

service:
  type: ClusterIP
  port: 80
```

The order

From weak to strong, so the lower one wins:

1. The chart's `values.yaml`.
2. Values from the parent chart, if it's a subchart.
3. Files with `-f`, the right one beats the left.
4. `--set` and its variants, the right one beats the left.

```
helm upgrade web ./web -f prod.yaml -f secrets.yaml --set
replicaCount=5
```

Memory aid: Read from left to right: Whoever speaks last wins.

> **PICTURE THIS**
>
> *Sticky notes on a plan: Each new layer goes over the old one. In the end you only see the top note.*

The siblings of `--set`

Flag	Effect
`--set a=1`	Type is guessed, here a number.
`--set-string a=1`	Always a string.
`--set-literal a=1,2`	The whole text is the value, with no special-character logic.
`--set-json 'a={"b":1}'`	Value as JSON.
`--set-file a=file.txt`	Value from the contents of a file.

Paths for nested values: `--set image.tag=2.7.0`. List items: `--set hosts[0]=a.example.com`.

How values merge

- **Maps** are merged.
- **Lists** are replaced completely, never extended.
- `null` deletes a default value.

> **PITFALL**
>
> You want to add an environment variable, so you write the `env` list in `prod.yaml`. All entries from `values.yaml` are now gone. If you need configuration that can be extended, use maps instead of lists.

During an upgrade

Flag	Effect
No flag	Chart defaults plus whatever you specify now.
`--reuse-values`	The release's old values plus your new ones. New chart defaults are missing.
`--reset-values`	Only the chart's defaults.
`--reset-then-reuse-values`	New defaults, then the old values on top, then your new ones.

> **REMEMBER**
>
> The safest approach is to supply all values from files on every upgrade. Then the desired state lives in Git, not in your head. Use `--set` only for things like an image tag from CI.

IN THE INTERVIEW

In what order do values apply?

The chart's `values.yaml` is the weakest. Above it come parent chart values, then files with `-f` from left to right, then `--set` from left to right.

IN THE INTERVIEW

Why is `--reuse-values` risky?

It takes the values of the last release instead of the new chart defaults. If a new chart brings a new value, that value is missing, and the template fails or renders empty.

KEY TAKEAWAYS

- `values.yaml`, then `-f`, then `--set`. Right wins.
- Maps merge, lists replace.
- Values in files in Git, not in your head.

8

The Built-In Objects

What every template knows without any help from you.

Object	Examples
`.Values`	All merged values.
`.Release`	`.Name`, `.Namespace`, `.Revision`, `.IsInstall`, `.IsUpgrade`, `.Service`
`.Chart`	Everything from `Chart.yaml`: `.Name`, `.Version`, `.AppVersion`
`.Capabilities`	`.KubeVersion`, `.APIVersions.Has "..."`
`.Template`	`.Name`, `.BasePath`
`.Files`	Access to other files in the chart.

templates/configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-config
  namespace: {{ .Release.Namespace }}
data:
  chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
  revision: {{ .Release.Revision | quote }}
```

Asking about cluster capabilities

```
{{- if .Capabilities.APIVersions.Has
    "gateway.networking.k8s.io/v1" }}
# Create an HTTPRoute
{{- end }}
```

> **PITFALL**
>
> With `helm template`, `.Capabilities` doesn't ask any cluster. It uses built-in defaults. With `--api-versions` and `--kube-version`, you specify them explicitly. Otherwise your CI may render something different from the cluster.

IN THE INTERVIEW

How do you make a chart work with several Kubernetes versions?

With `.Capabilities`: The template asks whether an API exists and renders accordingly. `kubeVersion` in `Chart.yaml` locks out unsupported versions.

KEY TAKEAWAYS

- `.Values`, `.Release`, `.Chart`, `.Capabilities`, `.Template`, `.Files`.
- `.Release.IsUpgrade` distinguishes install from upgrade.
- `helm template` doesn't know your cluster.

PART III

Thinking in Templates

Go templates, whitespace, scope, building blocks. The foundation of every interview.

9

Template Fundamentals

Double braces, dots, and pipes.

Helm uses Go's template language, extended with the **Sprig** library. Everything between double curly braces is an **action**.

```
name: {{ .Release.Name }}
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

The dot

The dot is the current context. At the top of a template, that's the root with all objects. So `.Values.image.tag` means: from here, go to `Values`, then `image`, then `tag`.

Pipes

As in the shell, a value flows from left to right through functions:

```
env: {{ .Values.env | default "prod" | upper | quote }}
```

The same thing without a pipe, read from the inside out:

```
env: {{ quote (upper (default "prod" .Values.env)) }}
```

The value from the pipe always ends up as the **last** argument. That's why `default` takes the fallback value first.

> **PICTURE THIS**
>
> *An assembly line: The value comes in on the left, each station works on it, and the result drops out on the right.*

Why `quote`?

YAML guesses types. Without quotes, `true`, `1.20`, and `on` are not strings.

PITFALL

For an environment variable, `value: {{ .Values.debug }}` produces an error, because Kubernetes requires a string there and YAML reads `true` as a boolean. Always add `| quote` for strings. Numbers such as ports, on the other hand, go without.

Rendering as a check

```
helm template web ./web --show-only templates/service.yaml
```

Memory aid: Show just this one file. `-s` is the short form.

```
helm template web ./web --set image.tag=2.7.0 --debug
```

Memory aid: `--debug` shows more and also outputs broken YAML so you can see the error.

IN THE INTERVIEW

What does the dot mean in a Helm template?

The current context. At the top level it's the root, with `.Values`, `.Release`, and so on. `range` and `with` change it.

KEY TAKEAWAYS

- Action, dot, pipe.
- The pipe value is the last argument.
- Strings always with `quote`.

10

Whitespace

YAML is picky. A minus sign saves you.

An action leaves its line in place, even if it outputs nothing. What's left stays behind as a blank line.

Syntax	Effect
`{{- ...`	Removes whitespace on the **left**, including the preceding newline.
`... -}}`	Removes whitespace on the **right**, including the following newline.

```
metadata:
  labels:
    {{- if .Values.team }}
    team: {{ .Values.team }}
    {{- end }}
```

The minus eats the newline before the action. That way no blank lines are left over.

> **PICTURE THIS**
>
> *The minus is a vacuum cleaner pointing in one direction. It sucks up all the spaces and newlines on that side until it hits text.*

> **PITFALL**
>
> `{{- .Values.name -}}` eats on both sides. If the next line follows right after it, that line sticks to the previous one, and the YAML is broken. Only put a minus on the right if you want that.

Indenting blocks

For whole structures, you use `toYaml` and `nindent`:

```
    spec:
      containers:
        - name: web
```

```
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
```

`nindent 12` starts with a newline and indents every line by 12 spaces. The number matches the block's indentation in the target file.

Function	Effect
`indent 4`	Indents every line.
`nindent 4`	Like `indent`, but with a newline in front.

IN THE INTERVIEW

Why use `nindent` instead of `indent`?

With `nindent`, the action can sit on the left, and `{{-` removes the newline and the indentation before it. The result starts cleanly on a new line, and the template stays readable.

KEY TAKEAWAYS

- `{{-` eats left, `-}}` eats right.
- `toYaml` plus `nindent` for blocks.
- Troubleshooting: `helm template --debug`.

11

if, with, range

Conditions, context switches, and loops.

if

```
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
# ...
{{- end }}
```

These are false: `false`, `0`, the empty string, `null`, and empty lists or maps. Everything else is true.

Comparisons are functions and come first: `eq`, `ne`, `lt`, `gt`, `and`, `or`, `not`.

```
{{- if and .Values.ingress.enabled (eq .Values.env "prod") }}
```

> **PITFALL**
>
> A value of `0` counts as false. If you use `default 3 .Values.replicas`, even a deliberately set 0 becomes 3. The same goes for `false`. In such cases, check with `hasKey` or `kindIs`.

with

`with` moves the dot and skips the block if the value is empty.

```
{{- with .Values.nodeSelector }}
nodeSelector:
  {{- toYaml . | nindent 2 }}
{{- end }}
```

This is the clean form for optional blocks.

range

`range` loops over lists and maps. The dot becomes the current element.

values.yaml

```
hosts:
  - a.example.com
  - b.example.com
env:
  LOG_LEVEL: info
  REGION: eu
```

```
rules:
{{- range .Values.hosts }}
  - host: {{ . | quote }}
{{- end }}
env:
{{- range $name, $value := .Values.env }}
  - name: {{ $name }}
    value: {{ $value | quote }}
{{- end }}
```

Maps run in alphabetical order. That keeps the output stable, and an upgrade sees no false changes.

IN THE INTERVIEW

What is the difference between `if` and `with`?

Both check for empty. `with` also sets the dot to the value. That saves typing, but inside the block you can only reach `.Release` through `$`.

KEY TAKEAWAYS

- Empty values, 0, and `false` are false.
- `with` for optional blocks.
- `range` changes the dot.

12

Variables and Scope

The dollar sign takes you back home.

Inside `range` and `with`, the dot points somewhere else. That's where this error comes from:

```
{{- range .Values.hosts }}
  host: {{ . }}.{{ .Release.Namespace }}  # Error
{{- end }}
```

Here the dot is a string, and a string has no `.Release`.

The solution: `$`

`$` always points to the root of the template.

```
{{- range .Values.hosts }}
  host: {{ . }}.{{ $.Release.Namespace }}
{{- end }}
```

> **PICTURE THIS**
>
> *The dot is your current location. The dollar sign is the home button on your GPS: No matter where you are, it takes you to the start.*

Your own variables

```
{{- $name := include "web.fullname" . }}
{{- range .Values.ports }}
  - name: {{ $name }}-{{ .name }}
    port: {{ .port }}
{{- end }}
```

`:=` creates a variable, `=` reassigns it.

IN THE INTERVIEW

What do you do when you need to access `.Release` inside `range`?

I use `$.Release`, because `$` points to the root of the template. Alternatively, I create a variable before the loop.

KEY TAKEAWAYS

- `range` and `with` change the dot.
- `$` is always the root.
- `:=` creates, `=` changes.

13

define, include, _helpers.tpl

Building blocks instead of copies.

Defining a building block

templates/_helpers.tpl

```
{{- define "web.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name
    | trunc 63 | trimSuffix "-" -}}
{{- end }}

{{- define "web.selectorLabels" -}}
app.kubernetes.io/name: {{ .Chart.Name }}
app.kubernetes.io/instance: {{ .Release.Name }}
{{- end }}

{{- define "web.labels" -}}
{{ include "web.selectorLabels" . }}
app.kubernetes.io/version: {{ .Chart.AppVersion | quote }}
app.kubernetes.io/managed-by: {{ .Release.Service }}
helm.sh/chart: {{ .Chart.Name }}-{{ .Chart.Version }}
{{- end }}
```

`trunc 63` shortens the name, because many Kubernetes names may be only 63 characters long.

Using a building block

```
metadata:
  name: {{ include "web.fullname" . }}
  labels:
    {{- include "web.labels" . | nindent 4 }}
spec:
  selector:
    matchLabels:
      {{- include "web.selectorLabels" . | nindent 6 }}
```

The dot at the end passes the context to the building block. If you forget it, the building block has nothing to work with.

`include` instead of `template`

`template` can't pass its output into a pipe. `include` can. That's why you practically always use `include`.

> **PICTURE THIS**
>
> *`define` is the recipe in the binder, `include` is the cooking. The dot at the end is the ingredients you hand over.*

> **PITFALL**
>
> Named templates are **global**. Two charts with the building block `fullname` overwrite each other. So always prefix the chart name: `web.fullname`.

> **PITFALL**
>
> Selector labels must never change, because a Deployment's `selector` is immutable. A version in `matchLabels` makes the next upgrade fail. That's why the code above separates `selectorLabels` and `labels`.

IN THE INTERVIEW

Why `include` instead of `template`?

`include` returns a string you can process further, for example with `nindent`. `template` writes directly to the output. That prevents indentation.

KEY TAKEAWAYS

- `define` in `_helpers.tpl`, `include` in the template.
- The dot passes the context along.
- Names with a chart prefix.
- Selector labels stay stable.

14

Sprig: The Key Functions

About twenty functions cover almost everything.

Function	Effect
`default "x" .V`	Fallback for an empty value.
`coalesce .A .B "x"`	First non-empty value.
`quote`, `squote`	Quotation marks.
`upper`, `lower`, `title`	Upper and lower case.
`trim`, `trimSuffix`, `trimPrefix`	Trimming.
`trunc 63`	Shortening.
`replace "a" "b"`	Replacing.
`printf "%s-%d" .A .B`	Formatting.
`toYaml`, `toJson`, `fromYaml`	Converting.
`b64enc`, `b64dec`	Base64 for Secrets.
`sha256sum`	Checksum.
`list`, `dict`	Build a list or map.
`hasKey`, `get`, `dig`	Look into maps.
`merge`, `mergeOverwrite`	Merge maps.
`ternary "a" "b" .Cond`	Short form of if, then, else.
`semverCompare ">=1.30" .V`	Compare versions.

The image tag pattern

```
image: "{{ .Values.image.repository }}:{{ .Values.image.tag
  | default .Chart.AppVersion }}"
```

The pattern for new Pods on new configuration

If only a ConfigMap changes, the Pods don't restart. A checksum in the Pod annotations solves that:

```
template:
  metadata:
```

```
    annotations:
      checksum/config: {{ include (print $.Template.BasePath
        "/configmap.yaml") . | sha256sum }}
```

If the ConfigMap changes, the checksum changes, and with it the Pod template. Kubernetes rolls out again.

> **PITFALL**
>
> `randAlphaNum` generates a new value on every render. So a random password in a Secret changes on every upgrade. Read the old value with `lookup`, see chapter 15.

> **REMEMBER**
>
> For security reasons, Helm removes a few Sprig functions, including `env` and `expandenv`. A chart shouldn't depend on the environment of the machine it's rendered on.

IN THE INTERVIEW

How do you restart Pods when a ConfigMap changes?

With a checksum annotation in the Pod template. If the ConfigMap changes, the hash changes and with it the Pod template. That triggers a rolling update.

KEY TAKEAWAYS

- `default`, `quote`, `toYaml`, `printf`, `trunc`.
- Checksum in the Pod template for config rollouts.
- Randomness in a template causes changes on every upgrade.

15

tpl, required, lookup

Three functions with depth.

required: fail early and loudly

```
host: {{ required "ingress.host is missing!"
  .Values.ingress.host }}
```

If the value is missing, rendering stops with your message. That beats an empty field that only shows up at runtime. For your own checks, there's `fail`:

```
{{- if and .Values.ha (lt (int .Values.replicas) 2) }}
{{- fail "ha needs at least 2 replicas" }}
{{- end }}
```

tpl: values as templates

Normally a value is just text. `tpl` renders it as a template. That lets users put expressions in values.

values.yaml

```
hostname: "{{ .Release.Name }}.example.com"
```

```
host: {{ tpl .Values.hostname . }}
```

It needs the context as its second argument. Without `tpl`, the braces would appear literally in the manifest.

lookup: checking the cluster

```
{{- $old := lookup "v1" "Secret" .Release.Namespace "web-db" }}
{{- $pw := randAlphaNum 24 | b64enc }}
{{- if $old }}
  {{- $pw = index $old.data "password" }}
{{- end }}
apiVersion: v1
kind: Secret
```

```
metadata:
  name: web-db
data:
  password: {{ $pw }}
```

That way a password, once generated, stays the same across upgrades.

Argument	Meaning
1	`apiVersion`, for example `v1`.
2	`kind`, for example `Secret`.
3	Namespace, empty for cluster-wide.
4	Name, empty for a list.

PITFALL

`lookup` returns an empty result with `helm template` and `--dry-run=client`, because Helm doesn't ask the cluster. Argo CD also renders without lookup. The template has to work with an empty result. `--dry-run=server` does ask the cluster.

IN THE INTERVIEW

What does `lookup` do, and what are its limits?

It reads objects from the cluster during rendering. I use it, for example, to keep existing passwords. With `helm template` it's empty, and the output suddenly depends on cluster state. So I use it sparingly.

KEY TAKEAWAYS

- `required` for mandatory values, `fail` for rules.
- `tpl` renders values as templates.
- `lookup` is empty with `helm template`.

16

Reading Files

Configuration as real files in the chart.

Long configurations don't belong in YAML strings. Put them in a folder next to `templates/`.

```
web/
├── config/
│   ├── app.conf
│   └── logging.conf
└── templates/
    └── configmap.yaml
```

templates/configmap.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "web.fullname" . }}
data:
  {{- (.Files.Glob "config/*").AsConfig | nindent 2 }}
```

Call	Effect
`.Files.Get "a.conf"`	Contents of a file as a string.
`.Files.Glob "config/*"`	Several files by pattern.
`.AsConfig`	Shapes the result for a ConfigMap.
`.AsSecrets`	Shapes it, Base64-encoded, for a Secret.
`.Files.Lines "a.txt"`	Line by line, good for `range`.

A single file with values in it:

```yaml
data:
  app.conf: |-
    {{- tpl (.Files.Get "config/app.conf") . | nindent 4 }}
```

PITFALL

Files in `templates/` aren't visible to `.Files`. The same goes for anything `.helmignore` excludes. If the file is in the wrong place, you silently get an empty string.

IN THE INTERVIEW

How do you get a large configuration file into a chart?

I put it in a folder next to `templates/` and load it into a ConfigMap with `.Files.Get` or `.Files.Glob`. If it needs values, I also render it with `tpl`.

KEY TAKEAWAYS

- `.Files.Get`, `.Files.Glob`, `.AsConfig`.
- Files in `templates/` can't be read.
- `tpl` plus `.Files.Get` gives you templates in files.

PART IV

Advanced Charts

Schema, subcharts, library charts, hooks, tests, and CRDs.

17

values.schema.json

A contract for your chart's values.

Otherwise, typos in values fail silently. `replicas` instead of `replicaCount` simply does nothing. A JSON Schema file next to `values.yaml` validates the values.

values.schema.json

```
{
  "$schema": "https://json-schema.org/draft-07/schema#",
  "type": "object",
  "required": ["image"],
  "additionalProperties": false,
  "properties": {
    "replicaCount": { "type": "integer", "minimum": 1 },
    "image": {
      "type": "object",
      "required": ["repository"],
      "properties": {
        "repository": { "type": "string" },
        "tag": { "type": "string" }
      }
    }
  }
}
```

Helm checks the schema during `install`, `upgrade`, `lint`, and `template`.

Rule	Catches
`type`	String instead of number.
`required`	Missing required values.
`enum`	Values outside a list.
`minimum`	Nonsensical numbers.
`additionalProperties: false`	Typos in key names.

PICTURE THIS

The schema is the bouncer. `required` in a template only notices something's missing once it's inside. The schema checks at the door, and it checks everything at once.

PITFALL

`additionalProperties: false` at the top level can reject values for subcharts. For a chart with subcharts, you have to allow their keys and `global` in the schema.

```
helm lint ./web --set replicaCount=abc
```

Memory aid: Set wrong values on purpose and see whether the schema catches them.

IN THE INTERVIEW

What is `values.schema.json` for?

It validates values against a JSON Schema before Helm renders. Wrong types, missing values, and typos show up immediately, with a clear message. It also documents the chart's interface.

KEY TAKEAWAYS

- Schema checks during `install`, `upgrade`, `lint`, `template`.
- `additionalProperties: false` catches typos.
- A schema is documentation that doesn't go stale.

18

Dependencies and Subcharts

Charts build on charts.

Declaring dependencies

Chart.yaml (excerpt)

```yaml
dependencies:
  - name: postgres
    version: "~15.2.0"
    repository: oci://registry.example.com/charts
    condition: postgres.enabled
  - name: common
    version: "1.x.x"
    repository: file://../common
    alias: base
```

Field	Effect
`condition`	Value that switches the subchart on or off.
`tags`	Group switch for several subcharts.
`alias`	A different name, also for using the same chart twice.
`import-values`	Pulls the child's values into the parent chart.

```
helm dependency update ./web
```

Memory aid: Resolve versions, write `Chart.lock`, download into `charts/`.

```
helm dependency build ./web
```

Memory aid: Build `charts/` exactly from `Chart.lock`. This is the command for CI.

```
helm dependency list ./web
```

Memory aid: What's declared, and what's there?

PICTURE THIS

`Chart.yaml` is the shopping list: "milk, any 1.5%." `Chart.lock` is the receipt: exactly this brand, exactly this carton. CI shops by the receipt.

Passing values to subcharts

A child's values sit under its name:

`values.yaml`

```yaml
postgres:
  enabled: true
  storage: 20Gi

global:
  registry: registry.example.com
```

The child sees only its own section, as `.Values.storage`. Only `global` is visible to all of them, as `.Values.global.registry`.

PITFALL

Values in `values.yaml` are static text. An expression like the release name isn't rendered there. If the child needs a value like that, it has to render it itself with `tpl`.

REMEMBER

Subcharts end up in one release together with the parent chart. Don't count on Helm waiting until the database is ready before the application starts. The application has to cope with that itself.

IN THE INTERVIEW

`dependency update` or `dependency build`?

`update` re-resolves the version ranges and writes `Chart.lock`. `build` takes the versions from `Chart.lock`. In CI I use `build` so that builds are reproducible.

IN THE INTERVIEW

How do you pass values to a subchart?

In the parent chart's values, under the subchart's name. If everyone needs something, it goes under `global`.

KEY TAKEAWAYS

- `Chart.yaml` is the wish, `Chart.lock` the decision.
- Values sit under the subchart's name.
- `global` is visible to all.
- `condition` switches subcharts.

19

Library Charts

Shared code with no objects of its own.

With 30 microservices, you have 30 nearly identical charts. A **library chart** contains only building blocks. It produces no objects and can't be installed.

`common/Chart.yaml`

```yaml
apiVersion: v2
name: common
type: library
version: 1.0.0
```

`common/templates/_deployment.tpl`

```
{{- define "common.deployment" -}}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  # ...
{{- end }}
```

A service pulls in the library as a dependency and uses it with one line:

`templates/deployment.yaml`

```
{{ include "common.deployment" . }}
```

> **PICTURE THIS**
>
> *An application chart is a finished piece of furniture. A library chart is the box of standard screws that all the furniture draws from.*

	Application	Library
`type`	`application`	`library`
Installable	Yes	No
Produces objects	Yes	Only via `include` in the consumer
Purpose	Ship an application	Share patterns

PITFALL

A change in the library affects all consumers. Version it strictly with SemVer and update services deliberately, not all at once.

IN THE INTERVIEW

When is a library chart worth it?

When many charts share the same patterns, such as labels, Deployments, or security contexts. Changes are central and versioned, instead of in 30 copies. For two charts, it's rarely worth the effort.

KEY TAKEAWAYS

- `type: library`, only `define`.
- Pull in via `dependencies` and `include`.
- Version strictly.

20

Hooks

Stepping in at specific points in the lifecycle.

A hook is a normal object with an annotation. Helm runs it at a specific point in time. Database migrations are the typical case.

`templates/migration.yaml`

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ include "web.fullname" . }}-migration
  annotations:
    "helm.sh/hook": pre-upgrade,pre-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": before-hook-creation
spec:
  backoffLimit: 1
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: migrate
          image: {{ .Values.migration.image | quote }}
          command: ["./migrate"]
```

The hook points

Hook	Runs
`pre-install`, `post-install`	Before and after installation.
`pre-upgrade`, `post-upgrade`	Before and after an upgrade.
`pre-rollback`, `post-rollback`	Before and after a rollback.
`pre-delete`, `post-delete`	Before and after deletion.
`test`	When `helm test` is called.

Order and cleanup

- `hook-weight`: Hooks run in ascending order. The value is a string.
- `hook-delete-policy`: When the object goes away.

Policy	Meaning
`before-hook-creation`	Delete the old object before the next run. The default.
`hook-succeeded`	Delete after success.
`hook-failed`	Delete after failure.

PICTURE THIS

Hooks are like the crew before and after a concert: They set up and tear down, but they aren't in the program. They aren't part of the release.

PITFALL

Hooks aren't part of the release. Helm doesn't manage them: `helm uninstall` doesn't delete them, and the 3-way merge doesn't see them. Without a `hook-delete-policy`, old Jobs pile up.

PITFALL

A `pre-install` Job that needs a Secret from the chart won't find it, because normal objects don't exist yet at that point. Make the Secret a hook itself, with a lower weight.

IN THE INTERVIEW

How do you run database migrations with Helm?

As a `pre-upgrade` Job with `backoffLimit` and a clear delete policy. If it fails, the upgrade fails before new Pods start. Important: The migration must be backward compatible so that a rollback works.

IN THE INTERVIEW

Are hooks part of the release?

No. Helm runs them and waits, but doesn't manage them like release objects. They need their own delete policy.

KEY TAKEAWAYS

- The `helm.sh/hook` annotation turns an object into a hook.
- Weight controls the order.
- Hooks sit outside the release.
- `helm install --no-hooks` skips them.

21

Chart Tests

Is the application really running after Helm is done?

A chart test is a Pod with a hook annotation. If it ends with exit code 0, the test has passed.

`templates/tests/connection.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ include "web.fullname" . }}-test
  annotations:
    "helm.sh/hook": test
spec:
  restartPolicy: Never
  containers:
    - name: curl
      image: curlimages/curl:8.11.0
      args:
        - --fail
        - >-
          http://{{ include "web.fullname" . }}:{{
          .Values.service.port }}/
```

```
helm test web -n demo --logs
```

Memory aid: Start the tests in the cluster and show their logs afterward.

Chart tests check the **running** release. For rendering, there are unit tests, see chapter 31.

Tool	Checks	Cluster?
`helm lint`	Structure and schema.	No
`helm-unittest`	Rendered output.	No
`helm test`	The running application.	Yes

IN THE INTERVIEW

How do you test a Helm chart?

In layers: `helm lint`, unit tests with `helm-unittest`, schema validation of the manifests, then install in a throwaway cluster and run `helm test`.

KEY TAKEAWAYS

- A test is a Pod with `"helm.sh/hook": test`.
- Exit code 0 means passed.
- `helm test` needs a running release.

22

CRDs

Helm's sore spot.

Helm treats files in the `crds/` folder specially:

Rule	Consequence
No templating	No access to values.
Installed before everything else	Objects in the chart can use the CRD right away.
Only on first install	`helm upgrade` never changes them.
Never deleted	`helm uninstall` leaves them in place.

Why this caution? CRDs apply cluster-wide. If a CRD is deleted, all of its custom resources disappear in all namespaces. So upgrading one chart could hit other people's data.

> **PICTURE THIS**
>
> *A CRD is a law for the whole country. A chart is a single citizen. That's why a citizen may introduce the law once, but can't amend or repeal it on the side.*

Ways to update

Way	Trade-off
Dedicated CRD chart	CRDs in `templates/` of a separate chart. Clean and versioned separately.
Directly with `kubectl apply`	Simple, but an extra step in the process.
An operator manages them	Some projects bring this along themselves.
`crds/` plus manual work	You have to apply new CRD versions yourself.

```
helm show crds ./web | kubectl apply --server-side -f -
```

Memory aid: Get the CRDs from the chart and apply them yourself. That's the manual way.

PITFALL

A CRD in `templates/` plus objects of that CRD in the same chart goes wrong, because Helm checks the objects before creating them and doesn't know the kind yet: `no matches for kind`. A separate chart helps here.

PITFALL

If CRDs are in `templates/`, `helm uninstall` deletes them too. That takes all custom resources with them. Protect them with `"helm.sh/resource-policy": keep`.

IN THE INTERVIEW

Why doesn't Helm update CRDs in the `crds/` folder?

CRDs apply cluster-wide, and changing or deleting them hits all custom resources. So Helm installs them only once. I do updates deliberately, often through a dedicated CRD chart.

KEY TAKEAWAYS

- `crds/`: installed once, never changed, never deleted.
- A dedicated CRD chart for clean updates.
- `resource-policy: keep` protects against deletion.

PART V

Operations

Upgrades, release storage, apply, Helm 4, registries, signatures, and Secrets.

23

install, upgrade, rollback

The one command for every deployment.

The standard command

```
helm upgrade --install web \
  oci://registry.example.com/charts/web --version 1.4.2 \
  --namespace prod --create-namespace \
  -f values/prod.yaml \
  --wait --timeout 5m \
  --rollback-on-failure
```

`upgrade --install` is **idempotent**. If the release exists, it's upgraded. If not, it's installed. That's why the same line works the first time and the hundredth time.

> **PICTURE THIS**
>
> *`upgrade --install` is like a light with a remote control: You press "on," whether or not the light is already burning. The result is always the same.*

Waiting

Without `--wait`, Helm reports success as soon as the API has accepted the objects. The Pods can still crash after that. The default is `hookOnly`, meaning it waits only for hooks.

Strategy	Effect
`--wait`	Short for `--wait=watcher`. Watches status with kstatus.
`--wait=hookOnly`	Default: Waits only for hooks.
`--wait=legacy`	The waiting behavior of Helm 3.
`--wait-for-jobs`	Also waits until Jobs are finished.

`--timeout` limits the wait. The default is five minutes.

> **PITFALL**
>
> Without a `readinessProbe`, a Pod is ready as soon as the process starts. Then `--wait` doesn't help much. Helm is only as good as your probes.

When things go wrong

`--rollback-on-failure` automatically rolls back a failed upgrade. On a first install, it removes the release again. The flag also turns on `--wait`. In Helm 3 it was called `--atomic`.

Rollback

```
helm history web -n prod
```

Memory aid: Which revision was the last good one?

```
helm rollback web 4 -n prod --wait
```

Memory aid: Back to the state of revision 4, as a new revision.

A rollback restores only manifests.

> **PITFALL**
>
> A rollback doesn't undo a database migration and doesn't change any data in PVCs. For schema changes, the old version has to be able to run with the new schema. Plan for that before the upgrade.

Look first

```
helm upgrade web ./web -n prod -f prod.yaml --dry-run=server
```

Memory aid: Simulate against the cluster without changing anything.

Mode	Cluster?
`--dry-run=client`	No contact, like `helm template`.
`--dry-run=server`	Asks the cluster, also for `lookup`.

```
helm diff upgrade web ./web -n prod -f prod.yaml
```

Memory aid: The `helm-diff` plugin shows what would change, like `git diff`.

IN THE INTERVIEW

What does a safe upgrade in production look like?

A fixed chart version pin, values from files in Git, a diff beforehand, `upgrade --install` with `--wait`, a timeout, and `--rollback-on-failure`. After that, `helm test` and a look at the monitoring.

IN THE INTERVIEW

What does `helm rollback` restore, and what doesn't it?

The manifests of an earlier revision. It doesn't restore data, PVCs, or database schemas. The rollback is itself a new revision.

KEY TAKEAWAYS

- `helm upgrade --install` is idempotent.
- `--wait` and `--rollback-on-failure` go together.
- `--dry-run=server` and `helm diff` before production.
- Rollback doesn't undo data.

24

Where Helm Stores Its Releases

Helm's memory is a Secret in the namespace.

Helm has no server. Its memory lives in the cluster, in the release's namespace.

```
kubectl get secrets -n demo -l owner=helm
```

Memory aid: The label shows all release Secrets.

```
NAME                          TYPE                 DATA  AGE
sh.helm.release.v1.web.v1  helm.sh/release.v1   1     2d
sh.helm.release.v1.web.v2  helm.sh/release.v1   1     1h
```

There's one Secret per revision. The name gives away the release and revision. Everything is inside: chart, values, rendered manifest, and status.

Looking inside

```
kubectl get secret sh.helm.release.v1.web.v2 -n demo \
  -o jsonpath='{.data.release}' \
  | base64 -d | base64 -d | gzip -d
```

`base64` twice, because Kubernetes encodes the gzip that Helm already encoded one more time.

Release status

Status	Meaning
`deployed`	The current, active revision.
`superseded`	An older, replaced revision.
`failed`	The revision failed.
`pending-install`, `pending-upgrade`, `pending-rollback`	An action is running, or it's stuck.
`uninstalling`, `uninstalled`	Being removed, or removed with `--keep-history`.

Limits

- A Secret can be at most 1 MiB. Very large charts run into this.
- `--history-max` limits the revisions, the default is 10. Older ones are deleted.
- `HELM_DRIVER` selects the storage: `secret` (default), `configmap`, `memory`, or `sql`.

> **PICTURE THIS**
>
> *The Secret is the logbook on board the ship, not in the harbor office. If you know the ship (the namespace), you'll find the whole voyage in it.*

> **PITFALL**
>
> Anyone who can read Secrets in the namespace can also read all of the release's values, including passwords from `--set`. They're only compressed, not encrypted. Deleting the namespace also removes the history.

IN THE INTERVIEW

Where does Helm store its state?

As a Secret of type `helm.sh/release.v1` in the release's namespace, one per revision. It contains the chart, values, and manifest. Why there? There's no server, and this way RBAC applies automatically.

KEY TAKEAWAYS

- One Secret per revision.
- `helm get` reads these Secrets.
- Only compressed, not encrypted.
- `--history-max` keeps the list short.

25

3-Way Merge and Server-Side Apply

How Helm decides what to change in the cluster.

The problem

Other things in the cluster change your objects too: an HPA, a sidecar injector, an admin with `kubectl edit`. Should Helm overwrite that?

Client-side: the 3-way merge

Helm compares three states:

```
old:  last manifest from the release
live: what's in the cluster right now
new:  what the chart renders now
```

Old	Live	New	Result
2	2	3	Chart changed: becomes 3.
2	5	2	Chart unchanged: live stays 5.
2	5	3	Both changed: chart wins, becomes 3.

That means fields the chart doesn't change stay as they are in the cluster. Helm 2 knew only two states and missed manual changes.

> **PICTURE THIS**
>
> *A Git merge for the cluster: common ancestor, your change, their change.*

Server-Side Apply

With **Server-Side Apply** (SSA), the API server does the math, not Helm. For each field, it remembers a **field manager**, that is, who owns the field. If Helm wants to change a field someone else owns, the API reports a conflict instead of silently overwriting it.

```
helm upgrade web ./web -n prod --force-conflicts
```

Memory aid: A deliberate decision: In a conflict, Helm takes over the field anyway.

What applies when?

Situation	Method in Helm 4
New installation	Server-Side Apply.
Upgrade, rollback	Same as the release so far (`--server-side=auto`).
Release from Helm 3	Stays client-side.
Explicitly chosen	`--server-side=true` or `false`.

Drift

Drift means the cluster deviates from the chart. Helm doesn't detect this on its own. Only `helm diff` or a GitOps tool makes drift visible.

> **PITFALL**
>
> `replicas` hard-coded in the chart plus an HPA: The two fight over the same field. Leave the field out of the template when an HPA is active. With SSA, this kind of fight often shows up as a conflict.

IN THE INTERVIEW

What is the 3-way merge?

Helm compares the last release manifest, the live state, and the new manifest. That preserves other people's changes to fields the chart doesn't touch, while chart changes still get through.

IN THE INTERVIEW

What does Server-Side Apply change?

The API server does the merging and tracks ownership per field. Conflicts become visible instead of being silently overwritten. Helm 4 uses this for new installations, and existing releases keep their method.

KEY TAKEAWAYS

- Client-side: old, live, new.
- SSA: field managers and conflicts.
- New installations in Helm 4: SSA.
- Helm doesn't detect drift on its own.

26

What Helm 4 Changes

The move from Helm 3, short and complete enough for the conversation.

Helm 4 came out in November 2025. Existing `v2` charts keep working. It mainly changes how things work around them.

Renamed flags

Helm 3	Helm 4
`--atomic`	`--rollback-on-failure`
`--force`	`--force-replace`

The old names still work, but they print a warning.

Breaking changes to check

- **Post-renderers** are plugins. A path to a program is no longer enough.
- `helm registry login` takes only the domain name, not a full URL.
- **Server-Side Apply** is the default for new installations.
- **Scripts** that parse Helm's output should be rechecked.

What's new

Topic	New feature
Waiting	`--wait` uses kstatus and shows more precisely what's holding things up.
Plugins	New system with an optional WebAssembly runtime.
Plugin types	CLI, getter, and post-renderer.
Template functions	Plugins can add their own functions.
OCI	Installation by digest (`@sha256:...`).
Values	Values can be spread across multiple YAML documents.
Cache	Content-based cache for charts.
SDK	A stable Go API.

REMEMBER

Since the special release of September 9, 2026, Helm 3 gets only security updates, and only until February 10, 2027. After that, it's over. Plan the move now.

The move in practice

1. Switch pipelines to the new flag names.
2. Check plugins. Installing `helm-unittest` under Helm 4 requires `--verify=false`.
3. Rebuild post-renderers as plugins.
4. On a test cluster, upgrade a Helm 3 release with Helm 4.
5. Check how Server-Side Apply affects fields that other tools manage.

IN THE INTERVIEW

What is the most important difference between Helm 3 and Helm 4?

Server-Side Apply as the default for new installations, waiting with kstatus, and a new plugin system with WebAssembly. Plus renamed flags such as `--rollback-on-failure`. Existing charts keep working.

IN THE INTERVIEW

Do Helm 3 releases keep working under Helm 4?

Yes. Helm 4 recognizes them, and upgrades keep the existing apply method. So they stay on client-side apply until you explicitly switch.

KEY TAKEAWAYS

- `--atomic` became `--rollback-on-failure`, `--force` became `--force-replace`.
- SSA for new installations.
- Post-renderers are now plugins.
- Helm 3 ends on February 10, 2027.

27

OCI Registries

Charts live where your images live.

An OCI registry stores not just images but arbitrary artifacts. Helm uses that for charts. You don't need a separate chart server or an `index.yaml`.

Publishing

```
helm registry login registry.example.com
```

Memory aid: Just the domain, without `oci://` and without a path.

```
helm package ./web
```

Memory aid: Produces `web-1.4.2.tgz`.

```
helm push web-1.4.2.tgz oci://registry.example.com/charts
```

Memory aid: Name and tag come from the package. You give only the folder.

Consuming

```
helm show chart oci://registry.example.com/charts/web --version 1.4.2
```

Memory aid: Read metadata without installing.

```
helm install web oci://registry.example.com/charts/web --version 1.4.2
```

Memory aid: The path replaces `repo/chart`. `helm repo add` isn't needed.

Pinning by digest

A tag can be overwritten, a digest can't.

```
helm install web \
  oci://registry.example.com/charts/web@sha256:abc123...
```

If the digest doesn't match, Helm doesn't install.

OCI or HTTP repository?

	OCI	HTTP repository
Infrastructure	Existing registry	Web server and `index.yaml`
Permissions	Same as for images	Your own solution
`helm repo add`	Not needed	Needed
`helm search repo`	Doesn't work	Works
Resolution by digest	Yes	No

PICTURE THIS

Charts used to have their own warehouse with their own catalog. Now they sit on the same shelf as the images, with the same inventory and the same keys.

PITFALL

`+` is allowed in SemVer but not in OCI tags. Helm replaces it with `_` in the tag. So a chart `1.0.0+build1` appears as `1.0.0_build1`.

IN THE INTERVIEW

Why charts in an OCI registry?

A single infrastructure for images and charts, with the same permissions, scans, and replication. I pin by digest and no longer need an index server.

KEY TAKEAWAYS

- `helm registry login`, `package`, `push`.
- `oci://` replaces `helm repo add`.
- A digest is immutable, a tag isn't.

28

Signing and Provenance

How do you know the chart is genuine?

A chart runs arbitrary code in your cluster. So you want to know who it came from and whether anyone has tampered with it.

Helm's own way: PGP

```
helm package ./web --sign --key 'Platform Team' --keyring
ring.gpg
```

Memory aid: `--key` is part of the name in the key, not a fingerprint.

This creates a file `web-1.4.2.tgz.prov` next to the package, with a checksum and signature.

```
helm verify web-1.4.2.tgz --keyring public.gpg
```

Memory aid: Verify authenticity, locally.

```
helm install web ./web-1.4.2.tgz --verify --keyring
public.gpg
```

Memory aid: Install only if verification succeeds.

The modern way: cosign

In OCI registries, a chart can be signed with `cosign` just like an image. The Sigstore project also allows keyless signatures using the CI identity.

```
cosign sign \
  registry.example.com/charts/web@sha256:abc123...
```

> **PICTURE THIS**
>
> *The digest proves the package is intact. The signature proves who packed it. For real trust, you need both.*

PITFALL

To sign, Helm needs the private key in a keyring file. If it's only in GnuPG 2's keystore, export it first: `gpg --export-secret-keys > ~/.gnupg/secring.gpg`. For verification, Helm 4.3.0 and later also reads GnuPG's keybox directly.

IN THE INTERVIEW

How do you secure the chart supply chain?

Charts from our own registry, pinned by digest, signed in CI, and verified before installation. Plus scans of the rendered manifests. I mirror third-party charts and read them first.

KEY TAKEAWAYS

- `.prov` plus `helm verify` for PGP.
- `cosign` for OCI.
- Digest for integrity, signature for origin.

29

Secrets

What must never be in a values.yaml in Git.

There are two leaks:

- **Git:** A password in `values.yaml` stays in the history forever.
- **Release:** Values from `--set` end up in the release Secret. `helm get values` shows them.

Base64 is not encryption.

The patterns

Way	Idea
`existingSecret`	The chart only references a Secret that's created some other way.
External Secrets Operator	A controller fetches data from Vault, AWS, or similar.
Sealed Secrets	An encrypted object sits in Git, and only the cluster can open it.
SOPS with `helm-secrets`	Encrypted values files in Git, decrypted at deployment.
CSI Secrets Store	The secret is mounted into the Pod as a file.

The `existingSecret` pattern

The chart should always allow a reference:

`values.yaml`

```yaml
database:
  existingSecret: ""
  passwordKey: password
```

`templates/deployment.yaml (excerpt)`

```yaml
env:
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: {{ required "database.existingSecret is missing"
```

```
          .Values.database.existingSecret }}
        key: {{ .Values.database.passwordKey }}
```

External Secrets

The chart ships only the request, not the secret:

templates/externalsecret.yaml

```yaml
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: {{ include "web.fullname" . }}-db
spec:
  secretStoreRef:
    name: vault
    kind: ClusterSecretStore
  target:
    name: {{ include "web.fullname" . }}-db
  data:
    - secretKey: password
      remoteRef:
        key: web/prod/db
        property: password
```

PICTURE THIS

The chart carries no key. All it has is a slip of paper saying which doorman is holding the key.

PITFALL

A Secret rendered with `{{ .Values.pw | b64enc }}` isn't safe. The value sits in Git or in the CI variable, in the release Secret, and in the manifest. And `helm template` in the CI log shows it too.

IN THE INTERVIEW

How do you handle Secrets in Helm?

No plaintext values in Git. Ideally, External Secrets fetches the data from a Vault, and the chart only references it. Without a Vault, use SOPS or Sealed Secrets.

KEY TAKEAWAYS

- Values end up in Git and in the release Secret.
- The chart references, it doesn't ship secrets.
- External Secrets, Sealed Secrets, SOPS.

PART VI

Quality and Delivery

Lint, unit tests, CI, GitOps, and the comparison with Kustomize.

30

Lint and Render

Find errors before a cluster sees them.

```
helm lint ./web --strict
```

Memory aid: **Lint** is what a lint roller picks up: the small stuff. `--strict` turns warnings into errors.

```
helm lint ./web --with-subcharts -f values/prod.yaml
```

Memory aid: Also check subcharts, with the real values.

> **REMEMBER**
>
> `helm lint` knows only chart rules. It doesn't check whether the result is valid Kubernetes. A typo like `replica:` instead of `replicas:` gets through.

Checking the result against Kubernetes

```
helm template web ./web -f values/prod.yaml | kubeconform -
strict -summary
```

Memory aid: Check the result against the Kubernetes schemas, without a cluster.

```
helm template web ./web | kubectl apply --dry-run=server -f -
```

Memory aid: Ask a real API server. This also catches admission rules.

Rendering every environment

A chart is only as good as the combination of its values. Check every environment in CI:

```
for env in dev staging prod; do
  helm template web ./web -f "values/$env.yaml" \
    | kubeconform -strict -summary
done
```

Checking security

Tools such as `trivy config`, `kube-linter`, or `checkov` find patterns like privileged containers or missing limits in the rendered YAML.

> **PICTURE THIS**
>
> *Lint checks the spelling, kubeconform the grammar, the dry run asks the building manager, and the security scan asks whether the door is open.*

IN THE INTERVIEW

What does `helm lint` check, and what doesn't it?

It checks structure, template syntax, and schema. It doesn't check whether the manifests are valid Kubernetes objects. For that, I render and check with `kubeconform` or a server dry run.

KEY TAKEAWAYS

- `helm lint --strict` is only the beginning.
- `helm template | kubeconform` checks the result.
- Render every environment.

31

helm-unittest

Tests for templates, without a cluster.

With lots of values and conditions, you can't check everything by hand. `helm-unittest` renders your templates and checks the result.

```
helm plugin install https://github.com/helm-unittest/helm-unittest.git --verify=false
```

Memory aid: Under Helm 4, this plugin needs `--verify=false`.

Tests live in `tests/` and end in `_test.yaml`.

tests/deployment_test.yaml

```yaml
suite: deployment
templates:
  - deployment.yaml
tests:
  - it: uses appVersion as the default tag
    release:
      name: web
    chart:
      appVersion: "2.7.0"
    asserts:
      - isKind:
          of: Deployment
      - equal:
          path: spec.template.spec.containers[0].image
          value: registry.example.com/web:2.7.0

  - it: sets the replicas from the values
    set:
      replicaCount: 5
    asserts:
      - equal:
          path: spec.replicas
          value: 5
```

```
helm unittest ./web
```

Memory aid: Runs all tests in the chart.

Important assertions

Assertion	Checks
`equal`	Value at a path.
`matchRegex`	Pattern at a path.
`contains`	Element in a list.
`notExists`, `exists`	Whether a path is present.
`hasDocuments`	Number of rendered documents.
`failedTemplate`	Whether rendering fails, for example with `required`.
`matchSnapshot`	Comparison with stored output.

What to test: the conditions. Every `if`, every `required`, and every default value deserves a test.

> **PITFALL**
>
> Snapshots pass even if the wrong thing was stored. If you just rewrite the snapshot on every difference, you're testing nothing. Use targeted assertions for what matters.

IN THE INTERVIEW

What do you test in a chart with unit tests?

Behavior: conditions on and off, default values, required values that must fail, and paths that should be absent in a configuration. Not every line of YAML.

KEY TAKEAWAYS

- Tests in `tests/*_test.yaml`.
- `set`, `asserts`, `equal`, `failedTemplate`.
- Test conditions, not every line.

32

CI Pipeline

From commit to signed chart.

The stages

1. **Lint:** `helm lint --strict`.
2. **Unit tests:** `helm unittest`.
3. **Render:** check every environment with `kubeconform`.
4. **Scan:** `trivy config` on the result.
5. **Integration:** `ct install` in a `kind` cluster.
6. **Publish:** on a tag, `package`, `push`, sign.

Chart Testing

The official tool `ct` (**chart-testing**) finds changed charts in a monorepo.

```
ct lint --target-branch main
```

Memory aid: Lint for changed charts. Also checks whether the version was bumped.

```
ct install --target-branch main
```

Memory aid: Installs them in a cluster and starts `helm test`.

An example

`.github/workflows/chart.yaml (core)`

```yaml
on:
  push:
    tags: ["v*"]

jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: azure/setup-helm@v5
      - run: helm dependency build ./web
      - run: helm lint ./web --strict
```

```
      - run: |
          helm package ./web \
            --version "${GITHUB_REF_NAME#v}"
      - run: |
          echo "$REG_PASSWORD" | helm registry login \
            registry.example.com -u ci --password-stdin
          helm push web-*.tgz \
            oci://registry.example.com/charts
        env:
          REG_PASSWORD: ${{ secrets.REG_PASSWORD }}
```

REMEMBER

Ideally, the pipeline also pins the Helm version and pins Actions by commit hash. Then the tool that builds your charts doesn't quietly change.

PITFALL

Pushing the same version to the registry twice overwrites the package, depending on the registry. A published chart version is immutable. If something changes, you bump the version.

IN THE INTERVIEW

What does a CI pipeline for charts look like?

Lint, unit tests, rendering all environments with schema validation, security scan, installation in `kind` with `helm test`. On a tag, package, push to the OCI registry, and sign.

KEY TAKEAWAYS

- Lint, test, render, scan, integration, publish.
- `ct` for monorepos.
- Chart versions are immutable.

33

GitOps: Argo CD and Flux

Git says what should run. A controller makes it happen.

With GitOps, nobody calls `helm upgrade` anymore. A controller in the cluster compares the desired state in Git with reality and reconciles the two.

Argo CD

application.yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: web
  namespace: argocd
spec:
  project: default
  source:
    repoURL: registry.example.com/charts
    chart: web
    targetRevision: 1.4.2
    helm:
      valueFiles:
        - values-prod.yaml
  destination:
    server: https://kubernetes.default.svc
    namespace: prod
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

Argo CD renders with `helm template` and applies the result itself. So Helm isn't managing a release. Since version 3.5, Argo CD renders only with Helm 4.

Flux

helmrelease.yaml

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2
kind: HelmRelease
metadata:
```

```
  name: web
  namespace: prod
spec:
  interval: 10m
  chart:
    spec:
      chart: web
      version: 1.4.2
      sourceRef:
        kind: HelmRepository
        name: company
  values:
    replicaCount: 3
```

Flux's Helm controller performs real Helm actions. Since Flux 2.8, it has used Helm 4: New releases run with Server-Side Apply, existing ones stay on client-side apply until you switch. The check with kstatus applies to all of them. For production, Flux recommends OCI charts with `OCIRepository` and `chartRef`.

The key difference

	Argo CD	Flux
How	Renders and applies itself	Performs Helm actions
Helm 4	From Argo CD 3.5	From Flux 2.8
`helm list`	Shows nothing	Shows the release
Hooks	Mapped to Argo hooks. Rollback and test hooks are ignored. If you define your own Argo hooks, Argo ignores all Helm hooks.	Run as in Helm
`lookup`	Returns empty	Works
Rollback	Through Git or the UI	Through Git, or automatically if remediation is configured

> **PICTURE THIS**
>
> *Argo CD reads the recipe and cooks it itself. Flux hands the recipe to the Helm chef and supervises the result. The dish is the same, but only the second one leaves a Helm entry in the kitchen log.*

PITFALL

Under Argo CD, `helm list`, `helm rollback`, and `lookup` come up empty. If you touch such a release by hand with Helm, you're fighting `selfHeal`. In an emergency: Fix it in Git.

IN THE INTERVIEW

How does Argo CD work with Helm?

It renders the chart with `helm template` and manages the objects itself. There's no Helm release in the cluster. Flux, on the other hand, performs real Helm releases. That matters for rollbacks and `lookup`.

KEY TAKEAWAYS

- Argo CD renders, Flux runs Helm.
- With GitOps, rollback means: revert in Git.
- `lookup` doesn't belong in Argo charts.

34

Helm or Kustomize?

Two paths to the same goal, often used together.

	Helm	Kustomize
Approach	Templates with variables	Plain YAML with patches
Language	Go templates	None, just YAML
Package	Versioned, distributable	No package format
Lifecycle	Release, history, rollback	None, just `kubectl apply -k`
Strength	Distributing software, many options	Adapting your own manifests per environment
Weakness	Templates quickly get unreadable	Unsuited to many users with many options

> **PICTURE THIS**
>
> *Helm is a form: The author decides what you can fill in. Kustomize is a highlighter: You can change any spot, but you have to know where it is.*

Rule of thumb

- You **publish** software for others: Helm.
- You adapt your **own** simple manifests: Kustomize.
- You use a **third-party chart** and need a field it doesn't offer: both.

Both together

A chart doesn't offer every option. Kustomize can patch the result afterward. Under Helm 4, that works through a post-renderer plugin:

```
helm upgrade web ./web --post-renderer my-kustomize-plugin
```

Memory aid: Under Helm 4, a plugin name, not a path to a script.

Alternatively, Kustomize renders the chart itself with the `helmCharts` field. That needs `--enable-helm` at build time.

IN THE INTERVIEW

When Helm, when Kustomize?

Helm when I want to package, version, and distribute software with many options. Kustomize for my own manifests with small differences per environment. For third-party charts with a missing option, I combine the two.

KEY TAKEAWAYS

- Helm: package and lifecycle.
- Kustomize: patches on YAML.
- Post-renderers connect the two.

PART VII

Troubleshooting

This is where the interview separates the wheat from the chaff.

35

The Method

First find the layer, then the error.

Every Helm error belongs to one of three layers:

Layer	Symptom	Tool
Render	Error with file name and line, before any contact with the cluster.	`helm template --debug`
API	The server rejects: validation, permissions, immutable fields.	`--dry-run=server`
Runtime	Everything accepted, but Pods aren't running.	`kubectl`, events, logs

> **REMEMBER**
>
> The first sentence in the interview is: "First I check which layer the error is in, and I don't change anything yet." Many Helm errors are really Kubernetes errors.

The first minute

```
helm status web -n prod
```

Memory aid: What state is the release in?

```
helm history web -n prod
```

Memory aid: What was the last good revision? The `DESCRIPTION` column often gives the reason.

```
helm get values web -n prod --all
```

Memory aid: With `--all`, all effective values, including the defaults.

```
helm get manifest web -n prod
```

Memory aid: What does Helm think is running?

```
kubectl get events -n prod --sort-by=.lastTimestamp
```

Memory aid: What does Kubernetes itself say?

```
helm upgrade web ./web -n prod --debug
```

Memory aid: `--debug` makes the output more verbose.

Common messages

Message	Chapter
`another operation ... is in progress`	36
`has no deployed releases`	36
`field is immutable`	37
`invalid ownership metadata`	38
`no matches for kind`	38
`conflict` during apply	38
`context deadline exceeded`	38

IN THE INTERVIEW

A Helm upgrade fails. What do you do?

I read the error message and classify it: render, API, or runtime. Then `helm status`, `helm history`, and for runtime errors, events and logs. I roll back only after I've captured the current state.

KEY TAKEAWAYS

- Render, API, runtime.
- `status`, `history`, `get values --all`, `get manifest`.
- Helm errors are often Kubernetes errors.

36

pending-upgrade

The classic: Helm thinks someone is still working.

```
Error: UPGRADE FAILED: another operation
(install/upgrade/rollback) is in progress
```

Why this happens

Before every action, Helm writes a new revision with status `pending-upgrade`. This entry acts like a lock. On SIGINT or SIGTERM, `helm upgrade` aborts in an orderly way and marks the revision as `failed`. But if Helm dies hard, the entry stays behind:

- The CI job was killed with SIGKILL.
- The laptop lost its connection.
- The runner's Pod was terminated hard before Helm finished.

> **PICTURE THIS**
>
> *A "Back in 5 minutes" sign on the door. The employee went home long ago, but the sign is still hanging there. Everyone is waiting outside.*

Fixing it

First check that nobody is really still working.

```
helm history web -n prod
```

Memory aid: Last line `pending-upgrade`? Which revision was `deployed` before it?

```
helm rollback web 7 -n prod
```

Memory aid: The clean way: Back to the last good revision.

If that doesn't work, there's a last resort:

```
kubectl get secrets -n prod -l owner=helm,status=pending-upgrade
```

Memory aid: Find exactly the Secret with the wrong status.

```
kubectl delete secret sh.helm.release.v1.web.v8 -n prod
```

Memory aid: Delete only this stuck revision. Then the previous one is the newest again.

> **PITFALL**
>
> Double-check the name first. Never delete the Secrets of all revisions. Without them, Helm no longer knows anything about the release and can't manage it. When in doubt, back up the Secrets first with `kubectl get secret -o yaml`.

Prevention

- Only one deployment per release at a time, built into CI as a lock.
- Set `--timeout` so that Helm gives up on its own before CI aborts it.
- Let CI jobs exit cleanly, don't kill them hard.

Failed first install

`has no deployed releases` means there are only failed revisions, no good one. So there's nowhere for a rollback to go.

```
helm uninstall web -n prod
```

Memory aid: Clean up and reinstall. Check first that no data is affected.

`--rollback-on-failure` on install prevents this, because Helm then cleans up after itself.

> **IN THE INTERVIEW**
>
> `another operation is in progress`**. What do you do?**
>
> I check that no process is really running. Then `helm history`, and I roll back to the last `deployed` revision. If that's not enough, I delete the stuck revision's Secret, and only that one. After that, I fix the cause in CI.

KEY TAKEAWAYS

- `pending-upgrade` is an orphaned lock.
- First `rollback`, then the specific Secret.
- Lock CI and set timeouts properly.

37

Immutable Fields

Some fields nobody may change, not even Helm.

```
Error: UPGRADE FAILED: ... spec.selector: Invalid value:
field is immutable
```

Kubernetes protects certain fields. Helm can only pass on what the API says.

Object	Immutable
Deployment, DaemonSet	`spec.selector`
StatefulSet	`selector`, `serviceName`, `volumeClaimTemplates`
Job	`spec.template`
Service	`clusterIP`
PVC	`storageClassName`, shrinking the size

The most common causes

- Labels in the selector have changed, for example through a new `fullnameOverride`.
- A version or a `helm.sh/chart` label is in `matchLabels`.
- The chart's major update rearranges the labels.
- A Job was deployed as a normal object and changed.

Fixing it

Case	Way
Deployment	Delete and recreate, or use a new object name. Plan for a brief interruption.
StatefulSet	`kubectl delete sts web --cascade=orphan`, then upgrade. Pods and PVCs stay.
Job	As a hook with `before-hook-creation`, or a name containing the revision.

```
kubectl delete statefulset web -n prod --cascade=orphan
```

Memory aid: An **orphan** has lost its parent: The object goes, its children stay.

PITFALL

`--force-replace` deletes and recreates objects. That causes an interruption, and for Services the IP may change. It's not a normal upgrade flag.

PICTURE THIS

The selector is the Deployment's house number. If you change it, it can't find its own residents anymore. That's why Kubernetes forbids it.

IN THE INTERVIEW

An upgrade fails with `field is immutable`. What's going on?

The chart wants to change a protected field, usually the selector. I use `helm diff` to see what changed. I undo the change, or I replace the object in a planned way, for StatefulSets with `--cascade=orphan`.

KEY TAKEAWAYS

- Selectors are immutable.
- No versions in `matchLabels`.
- `--cascade=orphan` saves StatefulSets.

38

Ownership and Conflicts

Who owns this object, and who may change it?

The object already exists

```
Error: ... exists and cannot be imported into the current
release: invalid ownership metadata
```

Helm manages only objects with these markers:

```yaml
metadata:
  labels:
    app.kubernetes.io/managed-by: Helm
  annotations:
    meta.helm.sh/release-name: web
    meta.helm.sh/release-namespace: prod
```

An object created by hand doesn't have them. You have two options:

- You set the markers with `kubectl label` and `kubectl annotate`.
- You use `--take-ownership`. Helm then takes over the object without this check.

> **PITFALL**
>
> Two releases that create the same object get in each other's way. This often happens with global objects such as ClusterRoles. Put the release name in the names of such objects.

Server-Side Apply conflicts

If a field belongs to another field manager, the apply fails. Find out who that is:

```
kubectl get deployment web -n prod -o yaml --show-managed-
fields
```

Memory aid: `managedFields` shows who owns which field.

Then you decide: Should the field come out of the chart, or should Helm take it over with `--force-conflicts`?

Removed APIs

```
Error: ... no matches for kind "Ingress"
in version "networking.k8s.io/v1beta1"
```

This happens after a Kubernetes upgrade: The stored manifest uses a removed API. Helm already fails when reading the old state. The `helm-mapkubeapis` plugin rewrites the release Secret to current APIs.

> **PITFALL**
>
> `helm-mapkubeapis` still builds on the Helm 3 SDK, and its docs don't mention Helm 4. Whether it runs reliably with Helm 4 is not established. Back up the release Secrets first and try it on a test cluster first.

> **REMEMBER**
>
> Before every Kubernetes upgrade, check your charts with `helm lint --kube-version` and with Pluto: `helm template ./web | pluto detect -`. That works regardless of the Helm version. `pluto detect-helm` checks running releases, but its docs mention only Helm 2 and 3.

Timeouts

`context deadline exceeded` means Helm waited, but something didn't finish. Helm is rarely to blame.

1. `helm status` and `kubectl get pods` show what isn't ready.
2. `kubectl describe` and logs explain why.
3. Typical causes: image can't be pulled, wrong probe, a hook Job is stuck, PVC stays `Pending`.

> **IN THE INTERVIEW**
>
> **`invalid ownership metadata`: What does that mean?**
>
> The object already exists but doesn't belong to this release. I check who created it. If Helm should take it over, I set the markers or use `--take-ownership`.

IN THE INTERVIEW

After a Kubernetes upgrade, every `helm upgrade` fails. Why?

The stored release manifest uses a removed API. The chart has to be updated, and the release itself can be rewritten with `helm-mapkubeapis`.

KEY TAKEAWAYS

- Helm recognizes ownership by label and annotations.
- `managedFields` shows conflicts.
- `mapkubeapis` repairs old APIs in the release.
- Timeout means: Look at the Pods.

PART VIII

The Interview

Questions, exercises, and a cheat sheet for the last evening.

39

The Questions

Short questions, short answers. For review before the conversation.

Fundamentals

What is Helm?

The package manager for Kubernetes: Templates plus values produce manifests, along with history and rollback.

Chart, release, repository?

Package, installed instance, storage location for packages.

What is a revision?

One version state of a release. Every `upgrade` and `rollback` creates a new one.

`version` or `appVersion`?

Chart version and application version, independent of each other.

Why is Tiller gone?

Broad permissions in the cluster. Today Helm uses the user's RBAC permissions.

Where does Helm run?

As a program on the client or in CI, not in the cluster.

Where does Helm store its state?

In one Secret per revision, in the release's namespace.

One chart several times in the same namespace?

Yes, with different release names, if the object names depend on them.

Values

In what order do values apply?

`values.yaml`, parent chart, `-f` from left to right, `--set`.

How are lists merged?

They aren't, they're replaced. Maps are merged.

How do you delete a default value?

Set it to `null`.

`--set` or `--set-string`?

`--set` guesses the type, `--set-string` forces a string.

Why is `--reuse-values` risky?

New chart defaults are missing.

What is `values.schema.json` for?

Validating values before rendering, including types and typos.

What does `global` mean?

Values that the parent chart and all subcharts can see.

Templates

What is the dot?

The current context.

What is `$`?

Always the root of the template.

`{{-` and `-}}`?

Remove whitespace on the left or right, including the newline.

`indent` or `nindent`?

`nindent` also adds a newline in front.

`include` or `template`?

`include` can be passed into a pipe.

`if` or `with`?

Both check for empty, `with` also changes the dot.

What does `required` do?

Stops with a message if the value is missing.

What does `tpl` do?

Renders a string as a template.

What does `lookup` do?

Reads objects from the cluster. With `helm template`, it's empty.

Why `quote`?

YAML guesses types, and strings should stay strings.

Why a prefix with `define`?

Named templates are global, even across subcharts.

Why `trunc 63`?

Many Kubernetes names may be only 63 characters long.

Why doesn't a ConfigMap change the Pods?

The Pod template stays the same. A checksum annotation solves that.

Why no random functions in Secrets?

Every run produces a new value, so there's a change on every upgrade.

Advanced charts

`dependency update` or `build`?

`update` resolves, `build` follows `Chart.lock`.

What is a library chart?

Building blocks only, not installable, `type: library`.

What is a hook?

An object with an annotation that runs at a point in the lifecycle.

Are hooks part of the release?

No. They need a delete policy.

How do you set the hook order?

With `helm.sh/hook-weight`, in ascending order.

What does `crds/` do?

Installs once, never changes or deletes.

How do you update CRDs?

A dedicated CRD chart, or deliberately with `kubectl apply`.

How do you protect an object during uninstall?

With `"helm.sh/resource-policy": keep`.

What is a chart test?

A Pod with `"helm.sh/hook": test`. Exit code 0 means passed.

Operations

Why `upgrade --install`?

Idempotent, same command for first install and update.

What does `--rollback-on-failure` do?

Rolls back or cleans up, formerly `--atomic`.

When is a release considered successful?

Without `--wait`, when the API has accepted the objects.

What does a rollback restore?

Manifests, not data.

3-way merge?

Comparison of old manifest, live state, and new manifest.

Server-Side Apply?

The API server does the merging and knows the owner of each field.

What's new in Helm 4?

SSA as the default for new installations, kstatus, new plugins, renamed flags.

Why OCI?

One registry for everything, same permissions, digests, no index.

How do you sign charts?

`helm package --sign` with PGP, or `cosign` in OCI.

How do you handle Secrets?

No plaintext. A reference in the chart, data from External Secrets, SOPS, or Sealed Secrets.

Argo CD and Helm?

Renders and applies itself. There's no Helm release.

Flux and Helm?

Performs real Helm actions.

Helm or Kustomize?

Helm for packages and lifecycle, Kustomize for patches to your own YAML.

When you don't know something

Say so. Then think out loud: "I haven't done it that way. I'd guess X, because Y. I'd check with `helm template` what comes out." A risk-free render is always a good answer with Helm.

KEY TAKEAWAYS

- Short answer first. Details only if someone asks.
- Mention what can go wrong in production.
- An example of your own beats any definition.

40

Exercises

Six exercises that often come up in interviews. With solutions.

> **REMEMBER**
>
> The flow in the interview: ask questions, plan out loud, start small, render, improve. Talk while you type.

Exercise 1: Optional Ingress

Situation: An Ingress should be created only if it's enabled, with any number of hosts.

`values.yaml`

```yaml
ingress:
  enabled: false
  hosts: []
```

`templates/ingress.yaml`

```yaml
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ include "web.fullname" . }}
spec:
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ . | quote }}
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: {{ include "web.fullname" $ }}
                port:
                  number: {{ $.Values.service.port }}
    {{- end }}
{{- end }}
```

What's happening here:

- `if` switches the whole object.
- `range` sets the dot to the host. That's why it's `$` in there.
- `include` gets `$`, because the building block needs the root.

Exercise 2: When a ConfigMap changes

Situation: Pods should restart automatically when the configuration changes.

Solution: A checksum as an annotation in the Pod template, see chapter 14. Add that this works only for values from the chart. If someone changes the ConfigMap by hand, nothing happens.

Exercise 3: Database migration without an outage

Situation: A new version needs a new database schema.

- **Hook:** `pre-upgrade` Job with `backoffLimit`, delete policy `before-hook-creation`.
- **Backward compatible:** add columns first, never delete in the same step. The old version must run with the new schema.
- **Cleanup:** Old columns go away only in the next release.
- **Rollback:** rolls back only manifests, never the schema.

Exercise 4: 30 microservices

Situation: All teams copy the same chart.

- **Library chart** with Deployment, Service, labels, and security defaults.
- Each service has a small chart with a `dependency` and its own values.
- Strict SemVer versions, automatic updates via Renovate or Dependabot.
- **Unit tests** for the library chart.
- Name the alternative: a single generic chart with values per service.

Exercise 5: Multiple environments

Situation: Dev, staging, and prod.

```
web/
├── Chart.yaml
├── values.yaml          shared defaults
└── values/
    ├── dev.yaml
    ├── staging.yaml
    └── prod.yaml
```

- Same chart, same version, different values.

- The same version moves through all stages. You don't rebuild per environment.
- Secrets never go in these files.
- Render and check every environment in CI.

Exercise 6: The upgrade is stuck in production

Situation: The pipeline was aborted, and a new upgrade reports `another operation is in progress`.

1. Make sure no other run is active.
2. `helm history`: Find the last good revision.
3. `kubectl get pods`: Is the application even still running? That takes priority.
4. `helm rollback` to the last good revision.
5. Only if that fails, delete the `pending` revision's Secret, after taking a backup.
6. Fix the cause: CI lock, timeouts, clean shutdown.

IN THE INTERVIEW

How would you design a chart for another team?

`values.yaml` with comments and safe defaults, a schema, clear error messages with `required`, the `existingSecret` pattern, examples in `README` and `NOTES.txt`, and unit tests for every option.

KEY TAKEAWAYS

- Start small and render.
- Don't forget `$` inside `range`.
- A rollback isn't the end of the story.

41

Cheat Sheet

Everything important in a few pages, for the last evening.

One line on everything

Term	Mnemonic
Chart	The recipe.
Release	The cooked dish.
`values.yaml`	Right wins.
Dot	Where you are right now.
`$`	The home button.
`{{-`	The vacuum cleaner.
Hook	The crew outside the program.
`crds/`	In once, never touched again.
`pending-upgrade`	The orphaned door sign.
Rollback	Manifests, not data.

Lifecycle

```
helm create web
helm template web ./web
helm lint ./web --strict
helm upgrade --install web ./web -n demo --create-namespace
helm list -A
helm status web -n demo
helm history web -n demo
helm rollback web 1 -n demo
helm uninstall web -n demo
```

Inspection

```
helm get values web -n demo --all
helm get manifest web -n demo
helm get hooks web -n demo
helm show values oci://registry.example.com/charts/web
```

```
helm template web ./web -s templates/service.yaml
helm template web ./web --debug
```

Packages

```
helm dependency update ./web
helm dependency build ./web
helm package ./web
helm registry login registry.example.com
helm push web-1.4.2.tgz oci://registry.example.com/charts
helm pull oci://registry.example.com/charts/web --untar
```

Template patterns

```
{{- with .Values.nodeSelector }}
nodeSelector:
  {{- toYaml . | nindent 2 }}
{{- end }}

labels:
  {{- include "web.labels" . | nindent 4 }}

image: "{{ .Values.image.repository }}:{{ .Values.image.tag
  | default .Chart.AppVersion }}"

host: {{ required "host is missing" .Values.host | quote }}

{{- range $k, $v := .Values.env }}
- name: {{ $k }}
  value: {{ $v | quote }}
{{- end }}
```

Flags you need to know

Flag	Meaning
`--wait`	Wait with kstatus (`watcher`).
`--timeout 5m`	Limit on the wait. Default: 5 minutes.
`--rollback-on-failure`	Formerly `--atomic`.
`--force-replace`	Formerly `--force`. Deletes and recreates.
`--force-conflicts`	Takes over fields with SSA.
`--server-side`	`true`, `false`, `auto`.
`--take-ownership`	Takes over existing objects.
`--dry-run=server`	Simulates against the cluster.
`--history-max`	Revisions to keep. Default: 10.
`--skip-crds`	Don't install CRDs.
`--no-hooks`	Don't run hooks.

Never without thinking

- Passwords in `values.yaml` in Git.
- Versions in `matchLabels`.
- `--reuse-values` in a pipeline.
- Deleting release Secrets without a backup.
- `--force-replace` as a default flag.
- Installing a chart without a pinned version.
- Random values without `lookup` in Secrets.

KEY TAKEAWAYS

- Render first.
- Values in Git, secrets outside.
- The cluster counts, not the template.

Index

$ 36
3-way merge 67
_helpers.tpl 38
appVersion 22
Argo CD 86
.Capabilities 27
Chart 12
.Chart 27
chart-testing 84
Chart.lock 50
Chart.yaml 20
checksum/config 41
cosign 73
CRD 59
crds/ 59
default 40
define 38
Drift 68
existingSecret 75
External Secrets 75
field is immutable 97
.Files 44
Flux 86
--force-conflicts 67
global 50
Helm 10
Helm 4 69
helm install 16
helm lint 80
helm test 57
helm upgrade 62
helm-diff 64
helm-unittest 82
helm.sh/hook 55
helm.sh/release.v1 65
helm.sh/resource-policy 60
HELM_DRIVER 65
.helmignore 20
HelmRelease 86
Hook 55
include 38
kind 14
kstatus 69
kubeconform 80
Kustomize 89
Library Chart 53
lookup 42
mapkubeapis 99
nindent 32
OCI 71
pending-upgrade 94
Pipeline 30
Post-Renderer 89
Provenance 73
quote 30
range 34
Release 12
.Release 27
Repository 12
required 42
Revision 12
--rollback-on-failure 62
Sealed Secrets 75
SemVer 22
Server-Side Apply 67
--set 24
SOPS 75
Sprig 40
Subchart 50
--take-ownership 99
Tiller 11
toYaml 32
tpl 42
--values 24
.Values 27
values.schema.json 48
values.yaml 20
--wait 62
with 34