Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 9, 2026

ReplicaSets and Deployments: Desired State, Self-Healing, and Scaling

Understand how Deployments and ReplicaSets turn a desired replica count into self-healing, replaceable Pods, then observe ownership and scaling with kubectl.

This is Learn post 06 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Earlier posts introduced API objects, labels and selectors, Pods, and container-level health. Now those ideas become a control loop that keeps an application at the size you declared.

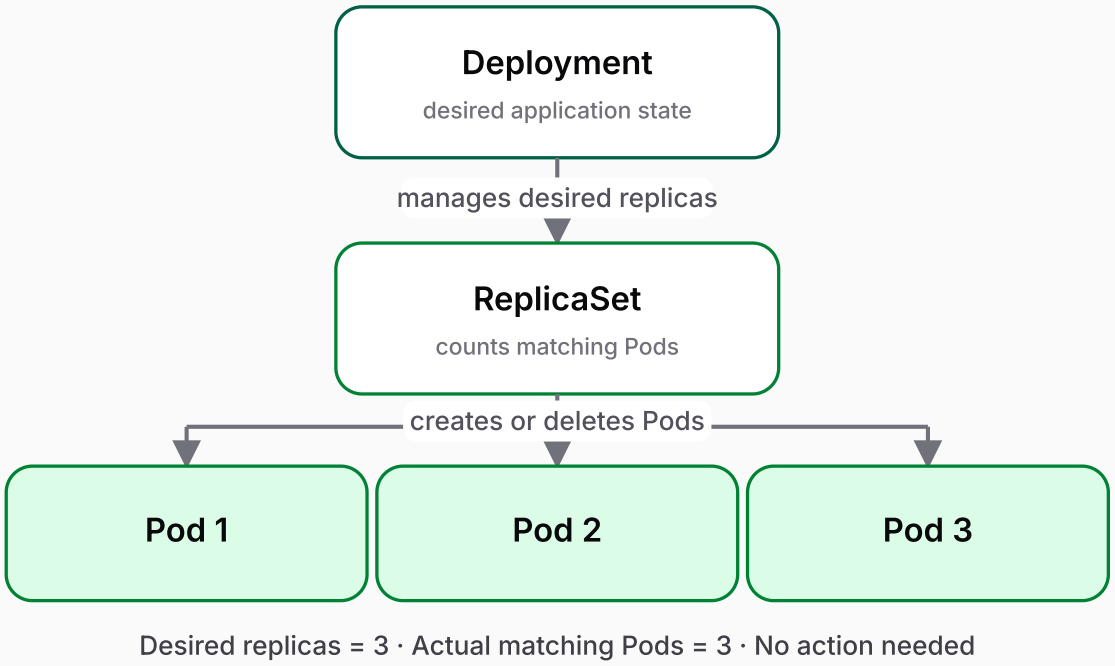
Administrators rarely want “start one Pod.” They want “keep three equivalent Pods running, replace lost ones, and let me change that number.” A ReplicaSet maintains the count; a Deployment is the normal object you manage above it.

What you'll learn

- Explain desired state and reconciliation without treating Kubernetes as a one-time command runner.
- Trace the ownership chain from a Deployment to a ReplicaSet to Pods.
- Read the selector, replica count, Pod template, and status fields that make the controller work.
- Observe Pod replacement and distinguish it from a container restart.
- Scale a Deployment imperatively and keep declarative configuration consistent.

The mental model: declare, compare, correct

A controller repeatedly compares desired state stored through the API with current cluster state. If they differ, the controller requests changes; if they match, it keeps watching. This is reconciliation.



Deployment to ReplicaSet to Pods

Think in layers: the Deployment owns application intent, the ReplicaSet enforces a replica count, and Pods are replaceable instances.

> Desired state is not a command that runs once. It is a condition Kubernetes keeps trying to make true.

Suppose desired replicas is three and one managed Pod disappears. Current becomes two. Because two is less than three, the ReplicaSet creates a new Pod from its template. The new Pod is a replacement with a new name and unique identifier (UID), not the old Pod repaired in place.

Deployment and ReplicaSet have different jobs

ReplicaSet: keep the count true

A ReplicaSet selects Pods by label and maintains a requested number of non-terminating Pods. It creates from a Pod template when there are too few and deletes managed Pods when there are too many.

You normally do not create or edit ReplicaSets directly. A Deployment creates and manages them for you. Direct ReplicaSets matter because they explain the counting and replacement behavior you observe beneath a Deployment.

Deployment: own the application intent

A Deployment stores the desired replica count and Pod template, then manages one or more ReplicaSets. For this lesson, focus on the current ReplicaSet and stable scaling. Changing the Pod template, controlled replacement, rollout status, and rollback are the subject of post 07.

- You normally apply, edit, scale, inspect, and delete the Deployment.
- The Deployment controller manages ReplicaSets.
- The ReplicaSet controller creates and deletes Pods.
- The scheduler and kubelet handle placement and container execution after a Pod exists.

Build one Deployment declaratively

Use the imperative generator when you want a valid starting manifest quickly. The client-side dry run prints an object without sending it to the API server.

bash

```
kubectl create deployment web \
  --image=nginx:1.27-alpine \
  --replicas=3 \
  --port=80 \
  --namespace=controllers-lab \
  --dry-run=client -o yaml > web-deployment.yaml
```

The cleaned manifest below keeps only the fields important to this lesson. The Deployment uses the stable apps/v1 API.

web-deployment.yaml · yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: controllers-lab
  labels:
    app: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.27-alpine
          ports:
            - name: http
              containerPort: 80
```

Read the manifest as a contract

- `spec.replicas: 3` is the desired number of Pods.
- `spec.selector` tells the controller which Pods belong in its set.
- `spec.template` is the blueprint used whenever a new Pod is required.

The selector and template labels must match. Here both say app=web. The API rejects a Deployment whose template cannot satisfy its own selector, and the selector is immutable after creation. Choose it deliberately.

Create the objects and observe every layer

Create an isolated namespace, apply the manifest, then list all three resource kinds together.

bash

```
kubectl create namespace controllers-lab
kubectl apply -f web-deployment.yaml
kubectl get deployment,replicaset,pod -n controllers-lab -l app=web
```

Representative output; generated suffixes vary · plaintext

```
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web   3/3     3            3           20s

NAME                        DESIRED   CURRENT   READY   AGE
replicaset.apps/web-6b4b85f75d   3         3         3       20s

NAME                  READY   STATUS    RESTARTS   AGE
pod/web-6b4b85f75d-8j9mt   1/1     Running   0          20s
pod/web-6b4b85f75d-kx7qs   1/1     Running   0          20s
pod/web-6b4b85f75d-vr2pn   1/1     Running   0          20s
```

One Deployment produced one ReplicaSet, which produced three Pods. The ReplicaSet name contains a hash derived from the Pod template, and each Pod name adds another generated suffix. Treat generated names as identities to discover, not strings to memorize.

In the Deployment row, READY is ready replicas over desired replicas. UP-TO-DATE reports replicas using the latest template, and AVAILABLE reports replicas that meet the Deployment availability rules. Post 07 uses those fields during an update; here, 3/3 and 3

available confirm steady state.

Ownership makes the hierarchy explicit

Labels let a controller find a set. Owner references record the managing relationship. Inspect both instead of inferring ownership from similar names.

```bash
kubectl get rs -n controllers-lab -l app=web \
  -o custom-columns='NAME:.metadata.name,OWNER-KIND:.metadata.ownerReferences[0].kind,OWNER:.metadata.ownerReferences[0].name,DESIRED:.spec.replicas'

kubectl get pods -n controllers-lab -l app=web \
  -o custom-columns='NAME:.metadata.name,OWNER-KIND:.metadata.ownerReferences[0].kind,OWNER:.metadata.ownerReferences[0].name'
```

```plaintext
NAME           OWNER-KIND   OWNER   DESIRED
web-6b4b85f75d   Deployment   web    3

NAME                OWNER-KIND   OWNER
web-6b4b85f75d-8j9mt   ReplicaSet  web-6b4b85f75d
web-6b4b85f75d-kx7qs   ReplicaSet  web-6b4b85f75d
web-6b4b85f75d-vr2pn   ReplicaSet  web-6b4b85f75d
```

The Deployment owns the ReplicaSet; the ReplicaSet owns the Pods. That chain also lets Kubernetes garbage collection identify dependents when an owner is deleted. Similar labels alone do not establish that deletion relationship.

Self-healing is reconciliation after loss

Record one managed Pod, delete it, and watch the set. This is a normal demonstration of controller behavior: deleting the Pod creates a mismatch that the ReplicaSet corrects.

```bash
pod=$(kubectl get pods -n controllers-lab -l app=web \
  -o jsonpath='{.items[0].metadata.name}')

kubectl get pod -n controllers-lab "$pod" \
  -o jsonpath='{.metadata.name}{"  "}{.metadata.uid}{"\n"}'

kubectl delete pod -n controllers-lab "$pod"
kubectl get pods -n controllers-lab -l app=web --watch
```

Representative watch sequence; order and timing vary · plaintext

```
NAME              READY  STATUS       RESTARTS  AGE
web-6b4b85f75d-8j9mt  1/1    Terminating  0         2m
web-6b4b85f75d-kx7qs  1/1    Running      0         2m
web-6b4b85f75d-vr2pn  1/1    Running      0         2m
web-6b4b85f75d-z4c6h  0/1    Pending      0         0s
web-6b4b85f75d-z4c6h  1/1    Running      0         3s
```

Because the deleted Pod stops counting toward the ReplicaSet, actual replicas fall below three. The controller creates a new Pod from the template. The replacement has a new suffix and UID; Kubernetes restores the declared capacity, not the identity or in-memory state of the deleted Pod.

Pod replacement is not a container restart

Post 05 showed the kubelet restarting a failed container inside the same Pod: the Pod UID remains and restartCount rises. Here the ReplicaSet replaces an entire Pod: the old UID disappears and a new Pod starts with its own container restart count. These are different recovery layers.

> Self-healing means restoring a declared condition when Kubernetes has enough information and capacity to do so. It does not mean repairing application bugs, recovering unpersisted data, or guaranteeing that a replacement can be scheduled.

Scaling changes one desired number

When immediate operational intent is clear, kubectl scale updates the Deployment replica count. The Deployment passes the new target to its ReplicaSet, which creates the missing Pods.

```bash
kubectl scale deployment/web -n controllers-lab --replicas=5
kubectl get deployment/web -n controllers-lab --watch
# Press Ctrl-C after READY reaches 5/5.
kubectl get deployment,replicaset -n controllers-lab -l app=web
```

Steady state after scaling up · plaintext

```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web   5/5     5            5           4m

NAME                           DESIRED   CURRENT   READY   AGE
replicaset.apps/web-6b4b85f75d   5         5         5       4m
```

Scaling up creates Pods from the same template. Scaling down deletes managed Pods until actual matches desired. Do not depend on a particular Pod surviving a scale-down; Pods in this workload are interchangeable controller-managed instances.

Keep the file and the live object aligned

kubectl scale changes the live object, but the manifest still says three. A later kubectl apply would restore three. If the intended declarative state is two, update the file and apply it:

web-deployment.yaml (edited fragment) · yaml

```yaml
spec:
  replicas: 2
```

```bash
kubectl apply -f web-deployment.yaml
kubectl get deployment,replicaset,pod -n controllers-lab -l app=web
```

Manual scaling changes a fixed target. Resource requests and limits are covered in post 10, while automatic replica decisions with the Horizontal Pod Autoscaler belong to post 14.

What the ReplicaSet counts—and what it does not

The ReplicaSet reconciles the number of selected, managed Pods. Readiness contributes to READY status, but an unready running Pod still exists and normally does not cause the ReplicaSet to create an extra replica. Readiness and Service routing are separate concerns.

If a matching bare Pod has no controlling owner reference, a ReplicaSet can adopt it. If that makes the set larger than desired, a Pod can then be deleted during reconciliation. Avoid reusing controller selectors casually across unrelated Pods.

The Pod template is used for new Pods; it does not continuously rewrite existing Pods. Editing a managed Pod is therefore not a durable way to change the workload. Put intended Pod configuration in the Deployment template. Post 07 explains what happens when that template changes.

Common mistakes and misconceptions

- Managing a Deployment-owned ReplicaSet directly. The Deployment is the source of application intent and may overwrite conflicting replica decisions.
- Deleting Pods as a way to scale down. The controller sees the loss and replaces them; change spec.replicas instead.

- Treating three Pod names as stable servers. ReplicaSets preserve a count, not Pod identity.
- Assuming Ready replicas and current replicas mean the same thing. A Pod may exist but not be ready.
- Using a selector that overlaps unrelated Pods. Selectors define controller membership and can lead to adoption or unintended reconciliation.
- Calling every restart “self-healing.” First identify whether the kubelet restarted a container or a controller replaced a Pod.

Administrator observation workflow

Start at the Deployment, move down the ownership chain, and read events where the controller records actions. This sequence answers whether desired and actual state differ and which layer acted.

```bash
kubectl get deployment -n <namespace> <name>
kubectl describe deployment -n <namespace> <name>
kubectl get rs -n <namespace> -l <selector> -o wide
kubectl describe rs -n <namespace> <replicaset>
kubectl get pods -n <namespace> -l <selector> -o wide
kubectl get events -n <namespace> --sort-by=.metadata.creationTimestamp
```

Use the Deployment selector you read from the object rather than guessing it from the name. Detailed rollout diagnosis returns in post 07, and the later Practice phase applies this workflow to broken workloads.

Clean up and continue

The namespace contains only this demonstration, so one command removes the Deployment and its dependents.

```bash
kubectl delete namespace controllers-lab
```

For current behavior and field details, keep the official Deployment documentation available.

Use the official ReplicaSet documentation when you need the lower-level controller semantics.

The controllers overview reinforces the desired-versus-current-state model.

The generated kubectl scale reference documents scaling syntax and preconditions.

What to remember

- A Deployment is the object you normally manage; it manages ReplicaSets, and ReplicaSets manage Pods.
- Reconciliation repeatedly compares desired state with current state and acts on any difference.
- spec.replicas is the target, spec.selector identifies the set, and spec.template creates replacements.
- ReplicaSets preserve a count of replaceable Pods, not individual Pod identity or application data.
- Container restart and Pod replacement are different recovery mechanisms at different layers.
- Scale the Deployment, and keep the declarative manifest aligned with the live desired state.