Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 10, 2026

Rolling Updates, Rollbacks, and Deployment Strategies

Understand how a Deployment replaces Pods safely through competing ReplicaSets, control availability with maxSurge and maxUnavailable, observe rollout progress, and roll back with confidence.



This is Learn post 07 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Post 06 established the ownership chain: a Deployment manages ReplicaSets, and ReplicaSets maintain replaceable Pods. Now we will use that chain to change an application without replacing every Pod at once.

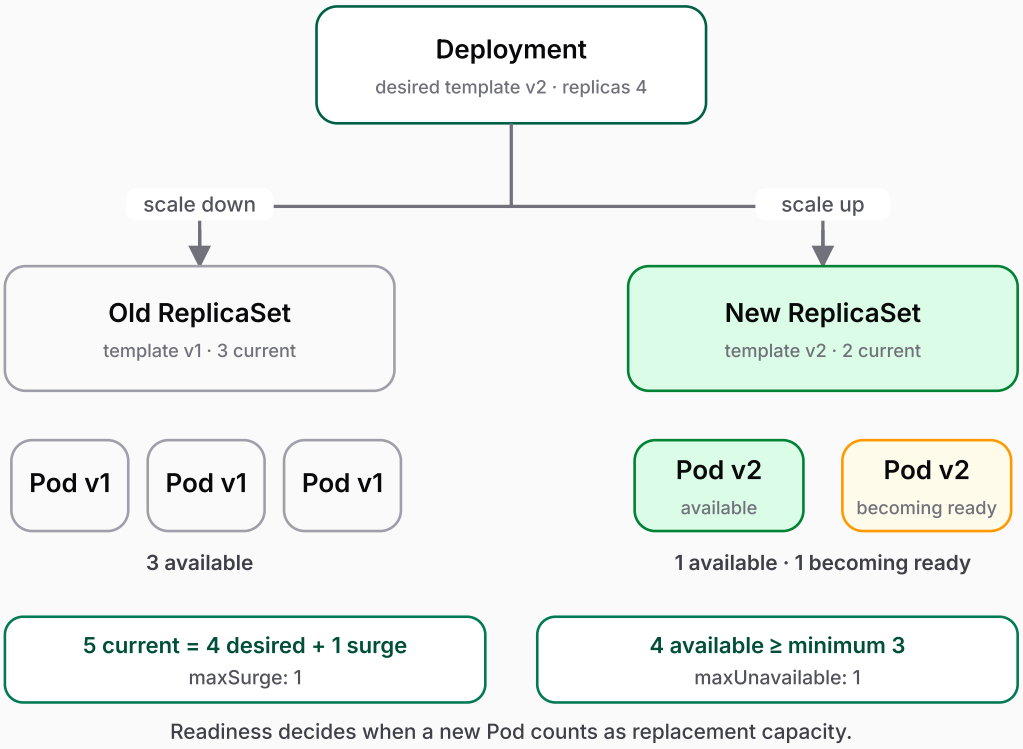
An update is not an in-place edit to running containers. Kubernetes creates Pods from a new template while retiring Pods created from the old template. The Deployment controls that exchange so capacity stays within limits you declare.

What you'll learn

- Explain a rolling update as coordinated scaling between an old and a new ReplicaSet.
- Use maxSurge, maxUnavailable, readiness, and minReadySeconds to reason about capacity during replacement.
- Start, observe, pause, resume, restart, and verify Deployment rollouts with kubectl.
- Inspect revision history and roll back to a known Pod template.
- Distinguish Kubernetes' native RollingUpdate and Recreate strategies from broader release patterns such as canary and blue-green delivery.

The mental model: two ReplicaSets exchange capacity

A Deployment revision represents a version of its Pod template. When that template changes, the Deployment creates or reuses the ReplicaSet for the new template. It then scales the new ReplicaSet up and the old ReplicaSet down.



A rolling update transfers capacity between ReplicaSets

The ReplicaSets do not update each other. The Deployment controller scales both while readiness determines when new Pods count as safe replacement capacity.

> A rolling update is controlled capacity transfer between ReplicaSets. Pods do not change version; old Pods disappear and new Pods replace them.

Because the Pod template is part of the ReplicaSet identity, a template change produces a different pod-template-hash label and therefore a different ReplicaSet. Changes outside the template, such as scaling spec.replicas, do not create a rollout revision.

Declare the replacement policy

The manifest matters here because the strategy is application policy, not a one-time command option. This Deployment runs four NGINX Pods and allows one extra Pod while permitting at most one desired replica to be unavailable.

web-deployment.yaml · yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: rollouts-lab
  labels:
    app: web
spec:
  replicas: 4
  revisionHistoryLimit: 5
  progressDeadlineSeconds: 120
  minReadySeconds: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.5-alpine
          ports:
            - name: http
              containerPort: 80
          readinessProbe:
            httpGet:
              path: /
              port: http
            initialDelaySeconds: 2
            periodSeconds: 2
```

A quick starting manifest can still be generated with `kubectl create deployment web --image=nginx:1.27.5-alpine --replicas=4 -n rollouts-lab --dry-run=client -o yaml`. Add the rollout policy and probe declaratively because those fields explain how replacement should proceed.

Read the four safety controls

- `maxUnavailable: 1` means at least three of the four desired replicas should remain available during the rollout.

- `maxSurge: 1` lets the controller raise the active replica target to five while creating replacements. Terminating Pods can make the visible Pod count temporarily higher until their grace periods end.
- `minReadySeconds: 5` requires a new Pod to remain ready for five seconds without a container crash before the Deployment considers it available.
- `progressDeadlineSeconds: 120` lets the Deployment report lack of progress after two minutes. Reporting a stalled rollout does not automatically roll it back.

Both maxSurge and maxUnavailable also accept percentages. Kubernetes rounds maxUnavailable down and maxSurge up; they cannot both resolve to zero. Absolute values make this four-replica demonstration easier to reason about.

Readiness controls the handoff

Post 05 introduced readiness probes. During a rollout, readiness becomes a replacement gate: because a new Pod does not count as available until it is ready and has satisfied minReadySeconds, the controller avoids scaling down old capacity too aggressively. A process that is running but not ready should not be treated as safe replacement capacity.

Create a stable starting revision

Create an isolated namespace, apply the manifest, and wait for its first rollout. rollout status watches until the Deployment completes or the command times out.

```bash
kubectl create namespace rollouts-lab
kubectl apply -f web-deployment.yaml
kubectl rollout status deployment/web -n rollouts-lab --timeout=2m
kubectl get deployment,replicaset,pod -n rollouts-lab -l app=web
```

Representative steady-state output; generated suffixes vary · plaintext

```
deployment "web" successfully rolled out

NAME               READY   UP-TO-DATE   AVAILABLE
deployment.apps/web   4/4     4          4

NAME                         DESIRED   CURRENT   READY
replicaset.apps/web-7d9c9b77f8   4         4         4

NAME                   READY   STATUS    RESTARTS
pod/web-7d9c9b77f8-2pw9j   1/1     Running   0
pod/web-7d9c9b77f8-6qx8b   1/1     Running   0
pod/web-7d9c9b77f8-jg5ft   1/1     Running   0
pod/web-7d9c9b77f8-vk2md   1/1     Running   0
```

READY 4/4 and AVAILABLE 4 establish a healthy baseline. UP-TO-DATE 4 means every desired replica uses the latest Pod template. One ReplicaSet currently holds all four replicas.

Change the Pod template and watch the exchange

kubectl set image is useful when the operational change is specifically a container image. It patches the image inside spec.template, so the Deployment observes a new Pod template and begins a rollout.

bash

```
kubectl set image deployment/web -n rollouts-lab \
  nginx=nginx:1.28.0-alpine

kubectl annotate deployment/web -n rollouts-lab \
  kubernetes.io/change-cause='Update nginx to 1.28.0-alpine' --overwrite

kubectl get deployment,replicaset,pod -n rollouts-lab \
  -l app=web --watch
```

The annotation gives the revision a human-readable reason in rollout history. The watch may skip intermediate states on a fast cluster, but the controller relationship remains the same.

One possible transition snapshot · plaintext

```
NAME                READY   UP-TO-DATE   AVAILABLE
deployment.apps/web   4/4     2          4

NAME                          DESIRED   CURRENT   READY
replicaset.apps/web-7d9c9b77f8   3        3        3
replicaset.apps/web-6f8b864c9c   2        2        1
```

The new ReplicaSet has two current Pods, but only one is ready. Three ready old Pods plus one ready new Pod preserve four available replicas. Five total current replicas use the allowed surge. As another new Pod becomes available, the controller can reduce the old ReplicaSet again.

Because controllers reconcile asynchronously, do not expect one fixed sequence of row values. Observe the direction: the new ReplicaSet moves toward four, the old ReplicaSet moves toward zero, and availability stays within policy when the cluster has enough capacity and the new Pods become ready.

Verify completion instead of assuming it

bash

```
kubectl rollout status deployment/web -n rollouts-lab --timeout=2m
kubectl get deployment/web -n rollouts-lab
kubectl get rs -n rollouts-lab -l app=web
kubectl get pods -n rollouts-lab -l app=web \
  -o custom-
columns='NAME:.metadata.name,IMAGE:.spec.containers[0].image,READY:.status.containerStatuses[0].ready,HASH:.metadata.labels.pod-template-hash'
```

Representative result after completion · plaintext

```
deployment "web" successfully rolled out

NAME                READY   UP-TO-DATE   AVAILABLE
deployment.apps/web   4/4     4           4

NAME                          DESIRED   CURRENT   READY
replicaset.apps/web-7d9c9b77f8   0         0         0
replicaset.apps/web-6f8b864c9c   4         4         4
```

Completion means all desired replicas are updated, available, and no old replicas remain running. The scaled-to-zero ReplicaSet is retained as revision history rather than deleted immediately.

Status tells you whether the controller is progressing

Use rollout status for a concise success or failure signal. Use describe when you need the Deployment conditions, ReplicaSet scaling events, and a reason for stalled progress. Then inspect the Pods owned by the new ReplicaSet.

bash

```
kubectl rollout status deployment/<name> -n <namespace> --timeout=2m
kubectl describe deployment/<name> -n <namespace>
kubectl get rs -n <namespace> -l <deployment-selector>
kubectl get pods -n <namespace> -l <deployment-selector> -o wide
kubectl get events -n <namespace> --sort-by=.metadata.creationTimestamp
```

A healthy Deployment normally reports Progressing and Available conditions. If no qualifying progress occurs before progressDeadlineSeconds, the controller reports ProgressDeadlineExceeded. That condition is evidence, not remediation: Kubernetes leaves the Deployment in place for an administrator or higher-level automation to handle.

Common causes include an image that cannot be pulled, failed readiness, insufficient cluster capacity for the surge, scheduling constraints, or runtime failures. Observability and systematic failure diagnosis are taught in posts 36 and 38; here the key is to follow the Deployment → ReplicaSet → Pod chain.

Revision history stores Pod templates

A Deployment keeps old ReplicaSets so their Pod templates can be inspected and reused. revisionHistoryLimit controls how many old ReplicaSets are retained after completed rollouts.

```bash
kubectl rollout history deployment/web -n rollouts-lab
kubectl rollout history deployment/web -n rollouts-lab --revision=2
```

```plaintext
deployment.apps/web
REVISION  CHANGE-CAUSE
1         <none>
2         Update nginx to 1.28.0-alpine
```

The detailed history view shows the revision's Pod template, including the image. It is more reliable than trying to remember which generated ReplicaSet hash represented which release.

> Revision history covers the Deployment's Pod template. It is not a snapshot of every object the application uses, and it does not restore application data.

For example, changing a separately managed ConfigMap does not become part of the Deployment revision. ConfigMaps and Secrets have their own lesson in post 09.

Rollback means make an old template desired again

A rollback does not revive terminated Pods. It changes the Deployment's desired Pod template to a previous revision, after which normal rolling-update reconciliation creates replacement Pods from that template.

```bash
kubectl rollout undo deployment/web -n rollouts-lab --to-revision=1
kubectl rollout status deployment/web -n rollouts-lab --timeout=2m
kubectl get pods -n rollouts-lab -l app=web \
  -o custom-columns='NAME:.metadata.name,IMAGE:.spec.containers[0].image,READY:.status.containerStatuses[0].ready'
```

The explicit --to-revision form is safest when you have inspected history. Without it, rollout undo selects the previous revision. Verify rollback completion just as you verify a forward update; a previous template can still fail in today's cluster conditions.

Pause when several template changes belong together

A paused Deployment accepts spec changes but does not start a new rollout for them. This lets an administrator accumulate related template updates and release them as one revision on resume.

```bash
kubectl rollout pause deployment/web -n rollouts-lab
kubectl set image deployment/web -n rollouts-lab nginx=nginx:1.28.0-alpine
kubectl set resources deployment/web -n rollouts-lab -c nginx \
  --requests=cpu=100m,memory=64Mi
kubectl rollout resume deployment/web -n rollouts-lab
kubectl rollout status deployment/web -n rollouts-lab --timeout=2m
```

The example shows the workflow, not a requirement to batch every change. A paused Deployment cannot be rolled back until it is resumed, so always verify whether spec.paused is true when a template edit appears not to move.

Restart uses the same rollout machinery

Sometimes the desired image is unchanged but every Pod needs replacement—for example, after an operational dependency changes. rollout restart patches the Pod template with a restart timestamp, which triggers a normal rolling replacement.

```bash
kubectl rollout restart deployment/web -n rollouts-lab
kubectl rollout status deployment/web -n rollouts-lab --timeout=2m
```

This is controlled Pod replacement, not a container restart inside an existing Pod. New Pods receive new names and unique identifiers while the Deployment strategy still limits surge and unavailability.

RollingUpdate and Recreate are the native strategies

- **RollingUpdate** gradually replaces old replicas and is the default. It supports maxSurge and maxUnavailable and is appropriate when old and new application versions can overlap.
- **Recreate** scales old replicas down before creating new replicas during an upgrade. It accepts downtime and is useful when old and new versions must not run together.

```yaml
spec:
  strategy:
    type: Recreate
```

Do not add rollingUpdate fields when type is Recreate. Also separate the strategy from application guarantees: RollingUpdate reduces disruption only when enough capacity exists and replacement Pods become ready; Recreate deliberately creates a gap during upgrade.

Canary and blue-green are broader release patterns

Canary exposes a small portion of capacity or traffic to a new version before wider promotion. **Blue-green** keeps separate old and new environments and switches traffic between them. Neither name is a valid value of Deployment spec.strategy.type.

These patterns normally require multiple workloads plus traffic selection or routing. Services, Ingress, and Gateway API appear later in posts 28, 31, and 32, so their traffic mechanics are intentionally deferred.

A reusable administrator workflow

For a planned Deployment change, use a short sequence that preserves both intent and evidence.

1. Establish the current image, readiness, replica counts, and rollout history.
2. Change the Deployment's Pod template using a manifest or a focused kubectl command.
3. Watch rollout status and compare the old and new ReplicaSets.
4. Verify the final Pod images and availability; do not stop at a successful patch response.
5. If the release is unacceptable, inspect history, select a known revision, undo, and verify again.

Clean up and continue

The namespace contains only this demonstration, so removing it also removes the Deployment, ReplicaSets, and Pods.

```bash
kubectl delete namespace rollouts-lab
```

Use the official Deployment documentation for current strategy fields, rollout behavior, status conditions, and revision cleanup semantics.

Keep the generated kubectl rollout reference available for status, history, undo, pause, resume, and restart syntax.

The kubectl set image reference documents the focused image-update command used in this lesson.