Gergely Risko

How do you rollback deployments in Kubernetes?

PUBLISHED IN OCTOBER 2019

Welcome to Bite-sized Kubernetes learning — a regular column on the most interesting questions that we see online and during our workshops answered by a Kubernetes expert.

> Today's answers are curated by Gergely Risko. Gergely is an instructor at Learnk8s.

If you wish to have your question featured on the next episode, please get in touch via email or you can tweet us at @learnk8s.

Did you miss the previous episodes? You can find them here.

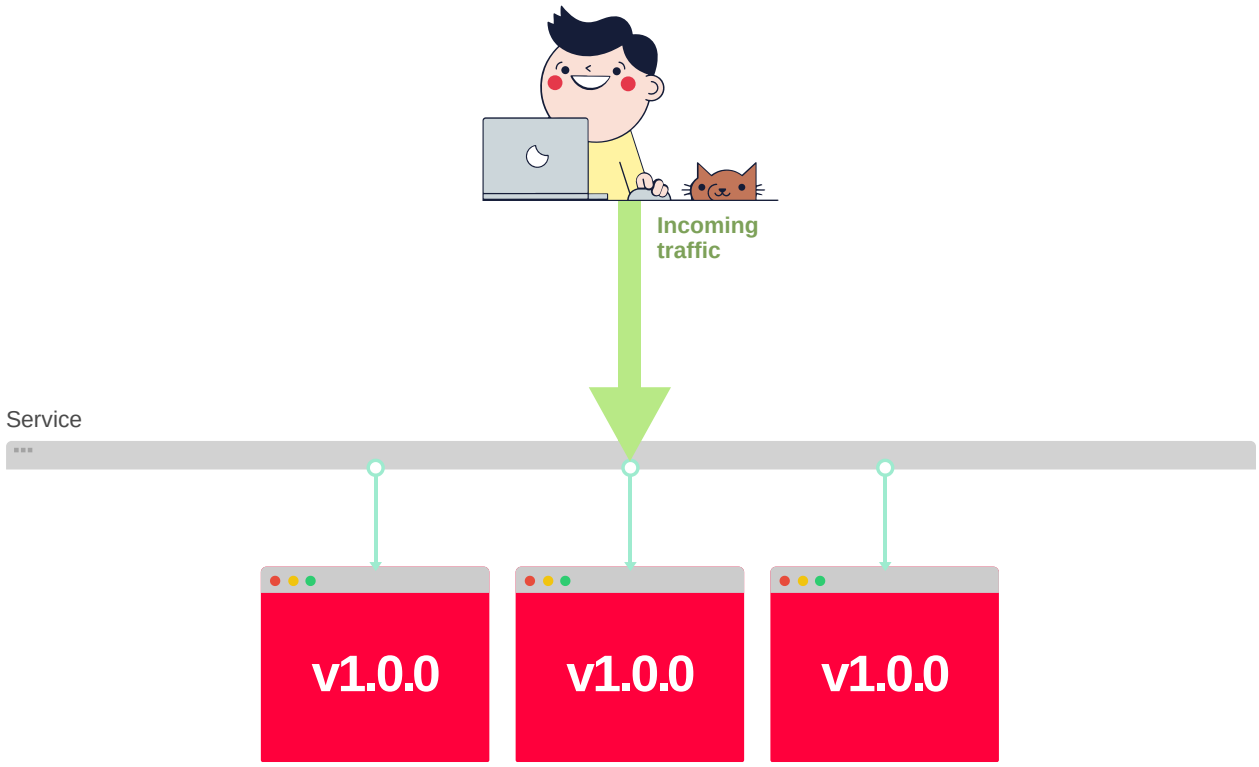
TL;DR: Kubernetes has a built-in rollback mechanism.

There are several strategies when it comes to deploying apps into production.

In Kubernetes, rolling updates are the default strategy to update the running version of your app.

The rolling update cycles previous Pod out and bring newer Pod in incrementally.

Let's have a look at an example:



1/16

NEXT ›

You have a Service and a Deployment with three replicas on version `1.0.0`. You change the `image` in your Deployment to version `2.0.0`, here's what happens next.

Zero-downtime deployment is convenient when you wish not to interrupt your live traffic.

You can deploy as many time as you want and your user won't be able to notice the difference.

However, even if you use techniques such as Rolling updates, there's still risk that your application doesn't work the way you expect it at the end of the deployment.

When you introduce a change that breaks production, you should have a plan to roll back that change

Kubernetes and `kubectl` offer a simple mechanism to roll back changes to resources such as Deployments, StatefulSets and DaemonSets.

But before talking about rollbacks, you should learn an important detail about Deployments.

You learned how Deployments are responsible for gradually rolling out new versions of your Pods without causing any downtime.

You are also familiar with the fact that Kubernetes watches over the number of replicas in your deployment.

If you asked for 5 Pods but have only 4, Kubernetes creates one more.

If you asked for 4 Pods, but you have 5, Kubernetes deletes one of the running Pods.

Since the replicas is a field in the Deployment, you might be tempted to conclude that is the Deployment's job to count the number of Pods and create or delete them.

This is not the case, unfortunately.

Deployments delegate counting Pods to another component: the ReplicaSet

Every time you create a Deployment, the deployment creates a ReplicaSet and delegates creating (and deleting) the Pods.

But why isn't the Deployment creating the Pods?

Why does it have to delegate that task to someone else?

Let's consider the following scenario.

You have a Deployment with a container on version 1 and three replicas.

You change the spec for your template and upgrade your container from version 1 to version 2.

The ReplicaSet can hold only a single type of Pod

So you can't have version 1 and version 2 of the Pods in the same ReplicaSet.

The Deployment knows that the two Pods can't coexist in the same ReplicaSet, so it creates a second ReplicaSet to hold version 2.

Then gradually it decreases the count of replicas from the previous ReplicaSet and increases the count on the current one until the latter ReplicaSet has all the Pods.

In other words, the sole responsibility for the ReplicaSet is to count Pods.

Instead, the Deployment manages ReplicaSets and orchestrates the rolling update.

But what if you don't care about rolling updates and only wish for your Pods to be recreated when they are deleted?

Could you create a ReplicaSet without a Deployment?

Of course, you can.

Here's an example of a ReplicaSet.

replicaset.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: example-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
```

For reference, this is a Deployment that creates the ReplicaSet above:

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
```

```
spec:
  replicas: 3
  selector:
    matchLabels:
      name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
```

Aren't those the same?

They are in this example.

However, in a Deployment, you can define properties such *how many Pods to create and destroy during a rolling update (the field is* `strategy` *).*

The same property isn't available in the ReplicaSet.

> How do you know which properties are available? You can consult the official API.

In general, the YAML for the Deployment contains the ReplicaSet plus some additional details.

You can create the ReplicaSet with:

bash

```
$ kubectl create -f replicaset.yaml _
```

There's something else worth noting about the ReplicaSets and Deployments.

When you upgrade your Pods from version 1 to version 2, the Deployment creates a new ReplicaSet and increases the count of replicas while the previous count goes to zero.

After the rolling update, the previous ReplicaSet is not deleted — not immediately at least.

Instead, it is kept around with a replicas count of 0.

If you try to execute another rolling update from version 2 to version 3, you might notice that at the end of the upgrade, you have two ReplicaSets with a count of 0.

Why are the previous ReplicaSets not deleted or garbage collected?

Imagine that the current version of the container introduces a regression.

You probably don't want to serve unhealthy responses to your users, so you might want to roll back to a previous version of your app.

If you still have an old ReplicaSet, perhaps you could scale the current replicas to zero and increment the previous ReplicaSet count.

In other words, **keeping the previous ReplicaSets around is a convenient mechanism to roll back to a previously working version of your app.**

By default Kubernetes stores the last 10 ReplicaSets and lets you roll back to any of them.

But you can change how many ReplicaSets should be retained by changing the `spec.revisionHistoryLimit` in your Deployment.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 3
  revisionHistoryLimit: 100
  selector:
    matchLabels:
      name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
```

What about the previous ReplicaSets?

Could you list all the previous Replicasets that belong to a Deployment?

You can use the following command to inspect the history of your Deployment:

```bash
$ kubectl rollout history deployment/app _
```

And you can rollback to a specific version with:

```bash
$ kubectl rollout undo deployment/app --to-revision=2 _
```

But how does the Deployment know their ReplicaSets?

Does it store the order in which ReplicaSets are created?

The ReplicaSets have random names with id such as `app-6ff88c4474` , so you should expect the Deployment to store a reference to them.

Let's inspect the Deployment with:

bash

```
$ kubectl get deployment app -o yaml _
```

Nothing is looking like a list of previous 10 ReplicaSets.

Deployments don't hold a reference to their ReplicaSets.

At least not in the same YAML.

Instead, related ReplicaSets are retrieved comparing the template section in YAML.

Remember when you learnt that Deployments are ReplicaSets with some extra features?

Kubernetes uses that information to compare Deployments and ReplicaSets and make sure that they are related.

What about the order?

How do you know which one was the last ReplicaSet used? Or the third?

Kubernetes stores the revision in the `ReplicaSet.metatada.annotation .`

You can inspect the revision with:

bash

```
$ kubectl get replicaset app-6ff88c4474 -o yaml _
```

In the case below the revision is 3:

replicaset.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: example-replicaset
  annotations:
    deployment.kubernetes.io/revision: "3"
spec:
  replicas: 3
  selector:
    matchLabels:
      name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
```

So, what happens when you find a regression in the current release and decide to rollback to version 2 like so:

bash

```
$ kubectl rollout undo deployment/app --to-revision=2 _
```

- Kubectl finds the ReplicaSets that belong to the Deployment

- Each ReplicaSet has a revision number. Revision 2 is selected
- The current replicas count is decreased, and the count is gradually increased in the ReplicaSet belonging to revision 2
- The `deployment.kubernetes.io/revision` annotation is updated. The current ReplicaSet changes from revision 2 to 4

If before the undo you had three ReplicaSets with revision 1, 2 and 3, now you should have 1, 3 and 4.

There's a missing entry in the history: the revision 2 that was promoted to 4.

There's also something else that looks useful but doesn't work quite right.

The history command displays two columns: *Revision* and *Change-Cause*.

bash

```
$ kubectl rollout history deployment/app
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>

$ _
```

While you're now familiar with the Revision column, you might be wondering what Change-Cause is used for — and why it's always set to `<none>`.

When you create a resource in Kubernetes, you can append the `--record` flag like so:

bash

```
$ kubectl create -f deployment.yaml --record _
```

When you do, Kubernetes adds an annotation to the resource with the command that generated it.

In the example above, the Deployment has the following metadata section:

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
  kubernetes.io/change-cause: kubectl create --filename=deployment.y
spec:
  replicas: 3
  selector:
    matchLabels:
      name: app
  template:
    metadata:
      labels:
        name: app
    spec:
      containers:
      - name: app
        image: learnk8s/hello:1.0.0
```

Now, if you try to display the history again, you might notice that the same annotation is used in the rollout history command:

bash

```
$ kubectl rollout history app
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deployment.yaml --record=true

$ _
```

If you change the container image in the YAML file and apply the new configuration with:

bash

```
$ kubectl apply -f deployment.yaml --record _
```

You should see the following new entry in the rollout history:

bash

```
$ kubectl rollout history deployment/app
REVISION  CHANGE-CAUSE
1         kubectl create --filename=deployment.yaml --record=true
2         kubectl apply --filename=deployment.yaml --record=true

$ _
```

The `--record` command can be used with any resource type, but the value is only used in Deployment, DaemonSet, and StatefulSet resources, i.e. resources that can be "rolled out" (see `kubectl rollout -h`).

But you should remember:

- The `--record` flag adds an annotation to the YAML resource, which can be changed at any time
- The rollout history command uses the value of this annotation to populate the Change-Cause table
- The annotation contains the last command only. If you create the resource and later use `kubectl scale --replicas=10 deploy/app --record` to scale it, only the scaling command is stored in the annotation.

Also, there is an ongoing discussion on deprecating the `--record` flag.

The feature provides little value for manual usage, but it still has some justification for automated processes as a simple form of auditing (keeping track of which commands caused which changes to a rollout).

That's all folks!

Advanced deployment strategies and rollbacks

If you enjoyed this article, you might find the following articles interesting:

- Architecting Kubernetes clusters — choosing a worker node size where you'll learn the pros and cons of having clusters with large and small instance types for your cluster nodes.
- Can you expose your microservices with an API gateway in Kubernetes? In Kubernetes, an Ingress is a component that routes the traffic from outside the cluster to your services and Pods inside the cluster. You can select an Ingress that is also an API gateway.