

— GUIDE · RABBITMQ

# RabbitMQ on Kubernetes (K8s) Troubleshooting Checklist: Broker or Platform?

Most RabbitMQ problems on Kubernetes that reach AceMQ support arrive described as a broker fault and turn out to be a platform one: storage, a security context, DNS, a disruption budget or a pod the operator is deliberately protecting. This checklist tells you which side you are on in five checks, then covers the six platform problems we see most. Operator behaviour checked against the RabbitMQ Cluster Operator documentation on 29 October 2026.



By **Scott Sternloff**, Senior Enterprise Architect

LinkedIn • Updated October 29, 2026 • 11 min read • 8 sections

— **BROKER OR PLATFORM: THE ANSWER IS USUALLY IN THE POD EVENTS**

Run `kubectl describe pod` before anything else. If the events show scheduling, image, volume or admission errors, it is the platform and no RabbitMQ setting will fix it. If the pods are running and ready, ask the broker: `rabbitmq-diagnostics check_running`, `alarms` and `cluster_status` inside a pod tell you whether RabbitMQ itself is unhappy. The one problem on this page that costs data rather than time is persistent volume retention: set the storage class reclaim policy to `Retain` and prove it in a scratch namespace before you rely on it.

## What this checklist is for

Use this page when RabbitMQ on Kubernetes is misbehaving and you need to know whether to look at the broker or the cluster it runs on. It is a triage list, not a deployment guide: the RabbitMQ on Kubernetes guide covers the operator, storage,

networking, resources and upgrades in full, and the blogs below compare the operator with Helm and the managed Kubernetes platforms. The items come from the six operator-related problems we see most in AceMQ support tickets, checked against the Cluster Operator documentation.

## Triage in five checks: broker or platform

Work these in order and stop at the first one that explains the symptom. The first three are platform checks; only the last two ask RabbitMQ.

- 1. Pod events.** `kubectl describe pod <pod> -n <ns>`. Insufficient CPU or memory, a wrong `imagePullSecrets` entry and a wrong `storageClassName` are the reasons the operator docs list for a cluster that never deploys. Any of these is the platform.
- 2. Cluster status conditions.** `kubectl describe rmq <cluster> -n <ns>`. `AllReplicasReady` and `ClusterAvailable` say whether pods are ready; `ReconcileSuccess` false means the operator's last reconcile failed and the cluster configuration may be out of date.
- 3. Operator logs.** `kubectl -n rabbitmq-system logs -l app.kubernetes.io/name=rabbitmq-cluster-operator`. An operator that cannot reach the Kubernetes API, or is blocked by a network policy, is a platform problem that looks like a RabbitMQ one.
- 4. Broker health inside a pod.** `kubectl exec` into a pod and run `rabbitmq-diagnostics check_running`, `rabbitmq-diagnostics alarms` and `rabbitmq-diagnostics cluster_status`. An alarm or a missing node here is the broker, and the troubleshooting decision tree takes over.
- 5. Quorum status.** `rabbitmq-diagnostics check_if_node_is_quorum_critical` on each pod. A quorum-critical pod explains a pod that will not terminate, and it is the check to run before any drain, upgrade or node replacement.

```
kubectl describe pod <pod> -n <ns>  
kubectl describe rmq <cluster> -n <ns>  
kubectl exec -n <ns> <pod> -- rabbitmq-diagnostics check_running  
kubectl exec -n <ns> <pod> -- rabbitmq-diagnostics check_if_node_is_quorum_critica
```

## Symptom to side: where each common problem lives

Most Kubernetes symptoms have one usual cause, and it is on the platform side more often than not. This table is the short version of the six problems below.

| SYMPTOM                                                                                                                      | USUALLY                                                        | FIRST CHECK                                                                        |
|------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|------------------------------------------------------------------------------------|
| <b>Pods Pending, never scheduled</b>                                                                                         | Platform: no default StorageClass, or not enough CPU or memory | Pod events; <code>kubectl get storageclasses</code>                                |
| <b><code>&lt;code&gt;ImagePullBackOff&lt;/code&gt;</code></b>                                                                | Platform: registry credentials                                 | <code>&lt;code&gt;imagePullSecrets&lt;/code&gt;</code> in the RabbitmqCluster spec |
| <b>Pods forbidden or failing at start under a security policy</b>                                                            | Platform: admission policy or security context                 | Replica set and StatefulSet events                                                 |
| <b>Pods restart with <code>&lt;code&gt;epmd&lt;/code&gt;</code></b><br><b><code>&lt;code&gt;nxdomain&lt;/code&gt;</code></b> | Platform: DNS caching or IPv6 resolution                       | CoreDNS cache setting                                                              |
| <b>Pods stuck terminating</b>                                                                                                | Broker, by design: the pod is quorum critical                  | <code>&lt;code&gt;check_if_node_is_quorum_critical&lt;/code&gt;</code>             |
| <b>Volumes gone after the cluster resource was deleted</b>                                                                   | Platform: reclaim policy                                       | <code>&lt;code&gt;persistentVolumeReclaimPolicy&lt;/code&gt;</code> on the PVs     |
| <b>Two brokers down at once during a node upgrade</b>                                                                        | Platform: disruption budget                                    | <code>&lt;code&gt;kubectl get pdb&lt;/code&gt;</code>                              |
| <b>Publishers blocked, pods healthy</b>                                                                                      | Broker: a memory or disk alarm                                 | <code>&lt;code&gt;rabbitmq-diagnostics alarms&lt;/code&gt;</code>                  |

**Table** • Common RabbitMQ on Kubernetes symptoms and which side they live on. Scroll sideways on small screens.

## Four setup decisions to confirm before you deploy or troubleshoot

Confirm the setup matches what the operator expects, because several problems below are a setup decision showing up later. The operator's defaults are modest: a RabbitmqCluster with no `replicas` set runs one node, so check what a new cluster actually deployed before you assume it is highly available.

- **Use the RabbitMQ Cluster Operator.** It is the RabbitMQ team's own operator and handles provisioning, reconciliation, scaling and rolling upgrades. If you hold a commercial licence, Tanzu RabbitMQ for Kubernetes is the supported path. The comparison with Helm charts is in the blog linked above.
- **Run RabbitMQ as the operator's StatefulSet, never a Deployment.** RabbitMQ nodes have stable identities: a node that comes back under a different name is a new node to the cluster, not the same one returning. The operator creates the StatefulSet for you.
- **Set an odd number of replicas: three, or five to survive two failures.** The operator docs say even numbers are highly discouraged. Kubernetes does not change the majority arithmetic.
- **Give production brokers dedicated nodes with affinity and tolerations.** In our tickets, a broker sharing a node with a noisy batch workload inherits its I/O contention and the symptoms look like broker faults. The RabbitmqCluster spec supports affinity rules and tolerations directly.

## The six platform problems that reach support

Each of these arrived at AceMQ support described as a RabbitMQ fault; none is a RabbitMQ bug. Each entry gives what happens and what to do.

- **1. Persistent volumes deleted with the cluster resource.** *What happens:* a RabbitmqCluster was deleted and its PVCs and data went with it, in one case despite a retain setting. *What to do:* set the StorageClass reclaim policy to

`Retain`, so the volume survives whatever happens to the claim, then delete a test cluster in a scratch namespace and confirm the volume is still there. The operator docs note that deleting a PVC deletes its data when `persistentVolumeReclaimPolicy` is `Delete`. Also note the operator's default volume is 10Gi, and with no default StorageClass the pods are never scheduled.

- **2. The container fails under an enforced pod security standard.** *What happens:* with non-root rules enforced, the container will not start or cannot write its data directory. *What to do:* set `runAsUser`, `runAsGroup` and `fsGroup` consistently with what the image expects, through the operator's `spec.override.statefulSet`. `fsGroup` is the one that matters for the data volume: without it the mounted volume is owned by root and the broker cannot write. On clusters with pod security policy admission, the operator docs say pods are forbidden until the policy and RBAC resources exist.
- **3. The operator fails to install without cert-manager.** *What happens:* installation fails on a webhook or certificate resource. *What to do:* the install guide says Cluster Operator 2.20 and later require cert-manager in the cluster. Install it first and wait for its pods to be ready before applying the operator manifest, not in the same apply.
- **4. epmd cannot resolve peers.** *What happens:* pods restart at boot with an `nxdomain` or `no_epmd_port` error. *What to do:* RabbitMQ nodes resolve their own and peer hostnames during boot, and the operator troubleshooting guide recommends lowering the CoreDNS cache from 30 seconds to 5 to 10. On IPv6-only clusters, which is where we have seen it most, Erlang distribution also has to be told to use IPv6 through the operator's Erlang INET configuration field. Fix resolution first; no RabbitMQ setting works around it.
- **5. Pods stuck terminating.** *What happens:* after deleting a cluster or scaling down, pods sit in Terminating with RabbitMQ still running. *What to do:* the operator's preStop hook holds any pod that is quorum critical, usually because of a leftover quorum queue, and the default termination grace period is 604,800 seconds, a full week. Check quorum status, make sure the queue's messages

are replicated elsewhere or acceptable to lose, and only then force-delete the pod. Forcing it earlier risks the data the hook is protecting.

- **6. Image pulls through a private registry.** *What happens:* pulls fail, or a mirrored image raises scanner warnings that look like vulnerability findings. *What to do:* relocate both the operator and RabbitMQ images to your registry as the install guide describes, reference the pull secret in the RabbitmqCluster spec in every namespace that runs brokers, and check the image digest against the upstream image rather than trusting the warning text. In our experience the warning is often about the mirror rather than the image.

## Upgrades and node maintenance

Most upgrade incidents on Kubernetes come from the disruption budget or from treating pod readiness as cluster health.

- **Set a PodDisruptionBudget with `maxUnavailable: 1`.** The operator docs give exactly this example. A percentage budget can round up: on a three-replica cluster, 50% allows two brokers down at once during a node upgrade, which loses the majority.
- **Check quorum status after each pod restarts, not just pod readiness.** The operator's readiness probe is a TCP check on the AMQP port (5672 by default), which the monitoring guide calls best practice because a probe that waits for a fully rejoined node can block a rolling restart. Ready therefore does not mean caught up: run the quorum-critical check before moving to the next pod.
- **Leave pod management at the operator's default.** The monitoring guide says the `OrderedReady` pod management policy is not recommended with RabbitMQ.
- **Upgrade the RabbitMQ image deliberately after an operator upgrade.** The operator does not change an existing RabbitmqCluster spec, so upgrading the

operator does not move clusters to a newer RabbitMQ version or new defaults. It also enables all feature flags at cluster start and cannot disable them.

- **Allow for the memory headroom the operator sets.** The operator sets RabbitMQ's usable memory 20% below the container limit, up to 2 GB, so the memory alarm fires earlier than the limit suggests. Size the limit with that in mind.
- **Follow the broker's own upgrade rules.** Version paths, feature flags and Erlang compatibility are the same on Kubernetes as anywhere else. Moving from 3.13 to 4.x still means migrating off classic mirrored queues first.

## Read the logs, the cluster status and the RabbitmqCluster metadata

When the triage in section 1 does not settle it, collect these before you delete or restart anything. Each shows a different layer, and the log you need depends on which side you are on.

| LAYER                       | COMMAND                                                                                              | WHAT IT SHOWS                                                                              |
|-----------------------------|------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Everything in the namespace | <code>kubectl -n &lt;ns&gt; get all</code>                                                           | Which pods, StatefulSets and services exist, and which are not ready.                      |
| Container log               | <code>kubectl logs &lt;pod&gt; -n &lt;ns&gt;</code> , adding <code>--previous</code> after a restart | What the container printed before it stopped, which is where a crash loop explains itself. |
| Broker log                  | <code>kubectl exec -n &lt;ns&gt; &lt;pod&gt; -- rabbitmq-diagnostics log_tail -N 200</code>          | The broker's own log, for cluster formation, authentication and alarm messages.            |
| Operator log                | <code>kubectl -n rabbitmq-system logs -l app.kubernetes.io/name=rabbitmq-cluster-operator</code>     | Why a reconcile failed, or why the operator cannot reach the Kubernetes API.               |

| LAYER          | COMMAND                                                         | WHAT IT SHOWS                                                    |
|----------------|-----------------------------------------------------------------|------------------------------------------------------------------|
| Cluster status | <code>kubectl describe rmq &lt;cluster&gt; -n &lt;ns&gt;</code> | The status conditions, including <code>ReconcileSuccess</code> . |

Then read the spec the operator is acting on, which is the RabbitmqCluster YAML manifest. The `apiVersion`, `kind` and `metadata` identify the object, and everything under `spec` is what you asked for. The values worth confirming on a cluster that misbehaves are replicas, the persistent volume and the resource limits. The operator docs give the defaults as one replica, a 10Gi volume, and memory and CPU limits of 2Gi and 2000m.

```
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
  name: production-rabbit      # ADAPT
  namespace: messaging        # ADAPT
spec:
  replicas: 3                  # odd: three, or five
  persistence:
    storageClassName: fast-ssd # ADAPT: must exist, or pods stay Pending
    storage: 50Gi              # ADAPT: size for a consumer outage
  resources:
    requests: { cpu: "2", memory: 4Gi } # ADAPT
    limits:   { cpu: "2", memory: 4Gi }  # ADAPT
```

Two cluster-formation notes sit here because they look like broker faults. The operator enables the Kubernetes peer discovery plugin and generates the cluster formation settings from the instance name, so you do not configure discovery by hand. And if you recreate a node, the docs warn that the procedure deletes the pod and its disk, so data not replicated elsewhere is lost; for a recreated `-0` node on RabbitMQ 4.1 and 4.2 the docs add that it must be told to join the existing cluster with `rabbitmqctl join_cluster rabbit@<cluster>-server-1` or it runs standalone, and a node deleted for good is removed with `rabbitmqctl forget_cluster_node`.

## **When it is still unclear which side it is**

Most of these take twenty minutes with someone who has seen them before. An AceMQ engineer joins a screen-share, reads the pod events and operator status with you, and tells you whether it is the broker or the platform. We reply within 1 hour to a request, and under a support contract a P1 gets a senior engineer within 15 minutes.

## **Frequently asked questions**

---

### **Why are my RabbitMQ pods stuck terminating on Kubernetes?**

The Cluster Operator's preStop hook holds any pod that is quorum critical, usually because a quorum queue would lose its majority if that pod went away, and the default grace period is a week. Check quorum status with `rabbitmq-diagnostics check_if_node_is_quorum_critical`, make sure the data is replicated or acceptable to lose, and only then force-delete.

---

### **Why do RabbitMQ pods restart with an `epmd nxdomain` error?**

RabbitMQ nodes resolve their own and their peers' hostnames during boot, and a stale DNS cache returns nothing. The operator troubleshooting guide recommends lowering the CoreDNS cache from 30 seconds to 5 to 10. On IPv6-only clusters, Erlang distribution also has to be configured for IPv6.

---

### **Does the RabbitMQ Cluster Operator need cert-manager?**

Yes, from Cluster Operator 2.20. The install guide says cert-manager must be installed in the cluster. Install it first and wait for its pods to be ready before applying the operator manifest.

---

## Will deleting a RabbitmqCluster delete my data?

It can. Whether the volumes survive depends on the persistent volume reclaim policy: with Delete, deleting the claim deletes the data. Set the StorageClass reclaim policy to Retain and test it by deleting a cluster in a scratch namespace before relying on it.

---

## What PodDisruptionBudget should RabbitMQ use?

maxUnavailable: 1, which is the example in the operator documentation. A percentage-based budget can round up and allow two of three brokers down at once during a node upgrade, which loses the cluster's majority.

---

## How do I get RabbitMQ logs on Kubernetes?

Start with `kubectl logs <pod> -n <namespace>`, adding `--previous` after a restart to see what the last container printed. For the broker's own log run `kubectl exec` into the pod and use `rabbitmq-diagnostics log_tail -N 200`. Operator logs are separate: `kubectl -n rabbitmq-system logs -l app.kubernetes.io/name=rabbitmq-cluster-operator`.

---

## How many replicas does a RabbitmqCluster get by default?

One. The operator docs give one replica, a 10Gi persistent volume and 2Gi memory and 2000m CPU limits as the defaults, so set replicas to three (or five) yourself if you want a cluster that survives a node failure.

---

## Related