

— GUIDE · RABBITMQ

RabbitMQ Production Readiness Checklist: 27 Checks Before You Deploy

A RabbitMQ cluster is ready for production when it has an odd number of nodes on quorum queues, alarm thresholds sized for the host, one user per application, a definitions export you can restore from, clients that confirm and acknowledge, and alerts that fire before publishing stops. This checklist is the 27 checks we run before calling a cluster production-ready, each with the reason it is on the list. Checked against rabbitmq.com on 29 October 2026, for RabbitMQ 4.4.



By **Scott Sternloff**, Senior Enterprise Architect

LinkedIn • Updated October 29, 2026 • 13 min read • 9 sections

— THE FOUR CHECKS THAT CATCH MOST ESTATES

If you have ten minutes, check four things: the cluster has an odd number of nodes, the data directories are excluded from antivirus and EDR scanning, definitions are exported on a schedule to somewhere off the node, and the alarm thresholds were set for this host rather than left as found. Any one of them turns a routine failure into an incident. On RabbitMQ 4.3 and later, also delete any leftover `cluster_partition_handling` line: the setting no longer does anything, and it misleads whoever reads the config during an incident.

How to use this checklist

Work top to bottom before go-live, or use it as an audit on a cluster you inherited, and treat any failed check as a blocker until someone has accepted the risk in writing. Every item carries a why, because a checklist without reasons gets followed

once and then ignored: nobody can tell which lines are load-bearing. If you disagree with an item, the why is the part to argue with.

This page is the pass/fail list. For the reasoning behind node counts, queue types and sizing, read the clustering and sizing guide; for TLS, permissions and CVE tracking in depth, the security and hardening guide. Each item comes from one of two places: the RabbitMQ documentation (production deployment guidelines, partitions, memory, Erlang versions) or a failure pattern from AceMQ support work, and the second kind is labelled as our practice.

Cluster topology and clustering for high availability (checks 1 to 5)

Topology is decided before any message is published, and most partition-related data loss traces back to a decision made here.

- **1. Run an odd number of nodes, three or five, spread across independent failure domains.** Three is the smallest cluster that tolerates one node failure and keeps a majority. Five tolerates two. Four still tolerates only one, and a clean two-and-two split leaves neither side a majority, which is why the RabbitMQ docs recommend odd counts.
- **2. On RabbitMQ 4.3 and later, remove `cluster_partition_handling` from `rabbitmq.conf`. On 4.2 and earlier, set it to `pause_minority`.** From 4.3.0 the metadata store is Khepri, Mnesia is gone, and the partitions guide says the partition-handling keys are accepted but have no effect: every replicated component is Raft-based and the majority side keeps serving. A leftover `autoheal` line misleads the next engineer. On older series the setting still applies, and `autoheal` or `ignore` let both halves keep taking writes, which is how divergence becomes data loss.
- **3. Use quorum queues for anything that must survive a node failure.** Quorum queues replicate with Raft and tolerate a minority of replicas failing by design. Classic queue mirroring was removed in RabbitMQ 4.0, so a 3.13 cluster still

relying on mirrored queues has to move to quorum queues before it can upgrade;

`rabbitmq-diagnostics`

`check_if_cluster_has_classic_queue_mirroring_policy` tells you whether it is.

- **4. Keep the inter-node link on a low-latency, reliable network.** Raft elections and inter-node traffic are sensitive to latency spikes, and in our experience a cluster stretched across regions produces partitions that look like application bugs. For cross-site traffic, use federation or shovel, which the docs describe as WAN-friendly, between separate clusters.
- **5. Run the same major Erlang/OTP version on every node, and check it against the compatibility matrix.** The Erlang guide highly recommends one major version across the cluster, and RabbitMQ refuses to cluster nodes it detects as potentially incompatible. RabbitMQ 4.4.0 supports Erlang 27.0 to 29.x and Erlang 26 is end of life. On 4.2.x and 4.3.0 to 4.3.5, moving a Khepri cluster from Erlang 27 to 28 has a known issue with mixed-version clusters, so plan that change with the RabbitMQ upgrade rather than as a rolling Erlang swap.

RabbitMQ node resources and alarms (checks 6 to 9)

Two resource alarms stop a broker accepting publishes, one on memory usage and one on free disk space, and these settings decide when they fire. The production guidelines also ask for at least 50,000 open file descriptors for the RabbitMQ user, so check `ulimit -n` while you are here.

- **6. Leave `vm_memory_high_watermark.relative` at the 0.6 default unless you have measured a reason to move it, and stay within 0.4 to 0.7.** The default blocks publishers at 60% of available memory. The docs call that a safe default, recommend 0.4 to 0.7, want at least 30% left to the OS and filesystem, and say values above 0.7 need solid memory monitoring. On nodes with quorum queues, budget at least three times the WAL size limit (512 MiB by default) on top of everything else.

- **7. Set `disk_free_limit` to at least the memory watermark, not the 50 MB default.** When the disk alarm fires, publishing stops on that node. The docs say the 50 MB default suits development only and recommend a free-disk limit about the size of the memory watermark: a node with a 4 GB watermark should have `disk_free_limit.absolute = 4G` as a minimum. Size the disk against a consumer outage, not steady state, or a small incident becomes a total one.
- **8. Exclude the RabbitMQ data directories from real-time antivirus and EDR scanning.** Scanning the metadata store, message stores, quorum queue WAL and segment files causes I/O stalls and file locks that throttle the node. This is AceMQ field practice rather than a line in the docs: it is the most common cause of unexplained latency we find on-premises, and the exclusion takes minutes.
- **9. Put the data directory on durable local storage, never on a distributed or shared filesystem.** The docs say quorum queues, streams and Khepri are designed for durable storage only, that distributed filesystems break the local `fsync` semantics RabbitMQ relies on, and that two nodes sharing a data directory is guaranteed corruption. NAS is acceptable only with low, stable latency; local SSD or NVMe is preferred.

Deploy-time baseline: node size, startup and routing

Five points sit underneath the numbered checks. They are easy to get right on day one and tedious to fix in a live cluster.

- **Size each RabbitMQ node for at least 4 CPU cores and 4 GiB of RAM.** That is the floor in the production guidelines, which also say RabbitMQ was not designed for single-core environments. It is a minimum, not a sizing result: throughput, queue count and retention decide the real number.
- **Seed the broker at startup, not by hand.** Definitions can be imported on node boot, so a replaced node comes back with the same vhosts, users and policies. The Erlang cookie is normally set by deployment automation, and the docs warn

that such tools can use default or placeholder values: confirm every node shares a cookie you generated.

- **Declare queues durable.** Khepri, quorum queues and streams are designed for durable storage only, and the deprecation list says non-durable, non-exclusive queues should be avoided. A transient queue loses its contents when the node restarts.
- **Know which queues are still classic queues.** Since 4.0 a classic queue is a single-replica queue: it is not mirrored, and all deliveries come from its leader replica. That suits transient or easily rebuilt workloads; anything that must survive a node failure belongs on a quorum queue (check 3).
- **Check that every exchange has a binding to a queue.** A message published to an exchange with no matching binding is dropped without an error unless the publisher sets `mandatory` or the exchange has an alternate exchange. Test the routing path end to end before go-live, not just the connection.

Access and identity (checks 10 to 13)

Access checks decide who connects, as whom, and whether you can tell afterwards.

- **10. Delete the default `guest` user before the cluster takes traffic.** The production guidelines say to delete it. `guest` can only connect from localhost because its credentials are well known, and a surprising number of deployments relax that restriction while debugging and never put it back. Create a separate administrative user with a generated password instead, and disable anonymous logins.
- **11. Give every application its own user and permissions, not one shared account.** Behind a load balancer every connection arrives from the same address, so the username is the only thing that tells you which application

opened thousands of channels. The docs recommend one user per application for exactly this reason, plus fine-grained permissions and credential rotation.

- **12. Separate tenants and environments by virtual host.** A virtual host (vhost) is the isolation boundary for exchanges, queues, permissions and limits. The docs recommend a vhost per tenant or environment in multi-tenant clusters; sharing one means one team's policy change lands on everyone.
- **13. Use TLS for client traffic, and verify the certificate name matches the hostname clients dial.** The docs recommend TLS with peer verification wherever possible, and TLS for inter-node and CLI traffic too. In our tickets, mismatched certificate names and pass-through load balancers produce connection failures that look like network faults. If TLS terminates at the balancer, the cluster link is still plaintext unless you configure inter-node TLS: decide that deliberately.

Backup and recovery (checks 14 to 17)

Recovery is the part that only gets tested during the incident unless you test it deliberately.

- **14. Export definitions on a schedule and archive them off the node.** Definitions carry vhosts, users, permissions, exchanges, queues, bindings and policies, and the backup guide calls a JSON export the recommended way to back them up. `rabbitmqctl export_definitions /path/file.json` works without the management plugin. Rebuilding definitions by hand under pressure is the slowest part of a recovery; a nightly export removes it.
- **15. Do not rely on snapshots of running RabbitMQ VMs as a backup.** The backup guide says backing up messages from under a running node is highly discouraged and can produce an inconsistent snapshot, that a node must be stopped to back up its messages, and that a restore only works on a node with exactly the same name. In our experience a restored snapshot brings back a

node that believes in a cluster state that no longer exists, and it has to be reset and rejoined before it is any use.

- **16. Write down the node recovery procedure and rehearse it once.** Stop, reset, rejoin, verify, in that order, with the traffic path blocked while you do it, then check quorum membership with `rabbitmq-queues quorum_status`. Teams that improvise this during an incident usually make the cluster state worse before it gets better.
- **17. Decide your disaster recovery position: federation, shovel, a standby cluster, or accepted loss.** Open-source RabbitMQ has no built-in cross-site replication; the backup guide points to separate standby clusters, and Tanzu RabbitMQ adds warm standby replication as a commercial feature. If nobody has chosen, the answer in practice is accepted loss, and it should be a decision rather than a discovery.

Client behaviour: publishers, consumers and acknowledgement (checks 18 to 23)

Most throughput problems sit on the client side of the connection rather than in the broker, so these six checks are about the applications.

- **18. Set a prefetch count. Do not leave it unlimited.** Unlimited prefetch lets one consumer pull a large share of the queue into memory and starve its peers. The docs say values from 100 to 300 usually give optimal throughput without overwhelming consumers, and quorum queues cap prefetch at 2,000 anyway. For slow per-message work, a prefetch of 1 or a small number keeps jobs spread across workers at some cost in throughput.
- **19. Acknowledge manually, after the work is done.** Automatic acknowledgement treats a message as handled the moment it is sent, so a consumer that dies mid-processing loses it. Manual acknowledgement after completion is what makes redelivery work.

- **20. Use publisher confirms for anything you cannot afford to lose.** Without confirms a publish is fire-and-forget: the broker can fail to route or store it and the publisher never learns. During a quorum queue leader election, confirms are delayed or come back as a nack, and only a publisher that waits for them knows to republish.
- **21. Keep connections long-lived, use separate connections for publishing and consuming, and close channels you no longer need.** Opening a connection per operation is expensive, causes connection churn and can exhaust file descriptors. The docs also recommend separate publisher and consumer connections so flow control on the publishing side does not slow consumer acknowledgements.
- **22. Keep heartbeats on, at a value below any load balancer or firewall idle timeout.** Without heartbeats, a client cut off by a firewall or NAT timeout leaves a connection the broker still believes is live, holding unacknowledged messages. The server suggests 60 seconds; the docs say 5 to 20 seconds is optimal for most environments and warn against values below 5.
- **23. Bound every queue with a length limit or TTL, and route overflow to a dead-letter exchange.** An unbounded queue behind a stalled consumer will reach the memory or disk alarm and stop publishing on the node. The bound decides what gets dropped instead of letting the broker decide for you, and the dead-letter exchange keeps the dropped messages visible.

Monitoring (checks 24 to 27)

The monitoring minimum is the one that lets someone answer "is it the broker?" without logging into it.

- **24. Scrape the `rabbitmq_prometheus` plugin on port 15692, not the management HTTP API.** The monitoring guide says the management plugin has more overhead, particularly in node memory, keeps only hours of history, and

ties the monitoring system to the system being monitored. The Prometheus endpoint returns aggregated metrics by default; per-queue alerts need the `/metrics/per-object` or `/metrics/detailed` endpoint.

- **25. Alert on queue depth trend, consumers dropping below the expected count, and unacknowledged messages.** Depth alone is noisy. Depth rising while a queue has no consumers is an actual incident, and it is the earliest reliable signal.
- **26. Alert on memory and disk headroom and on node availability as separate rules.** Memory and disk alarms block publishers without stopping the node, so a liveness check alone reports everything healthy while nothing is flowing.
- **27. Point the load balancer health check at a broker check, not a TCP open.** A node in an alarm state still accepts TCP connections and fails every publish. The management API's `/api/health/checks/local-alarms` and `/api/health/checks/ready-to-serve-clients` endpoints return 503 when the node cannot do work, and cost less per probe than running the CLI, which joins Erlang distribution on every call.

What this checklist does not cover

Two things sit outside this list on purpose. It does not size the cluster: node count, memory and disk depend on throughput and retention, and that is worked out in the sizing guide or during an implementation. And it does not tell you whether your version still receives security patches: RabbitMQ 4.4.0 is current as of 20 October 2026, with community support to 31 July 2027, and anything older needs checking against the release table.

If you want this run against your cluster, an AceMQ engineer walks the list with you on a screen-share, marks what is already covered and writes up the rest. We reply within 1 hour to a request.

Frequently asked questions

What should be on a RabbitMQ production checklist?

Cluster topology (odd node count, quorum queues, one Erlang major version), node resources (memory watermark, disk free limit, file descriptors, storage), access (no guest user, one user per application, vhosts, TLS), recovery (scheduled definitions export, a rehearsed node recovery, a decided disaster recovery position), client behaviour (prefetch, manual acks, publisher confirms, heartbeats, queue bounds) and monitoring (Prometheus plugin, trend alerts, a real health check).

Should I still set `pause_minority` on RabbitMQ 4.x?

Not on 4.3 or later. RabbitMQ 4.3.0 removed Mnesia and the partition-handling strategies; the `cluster_partition_handling` key is accepted but has no effect, and the docs say to remove it. On 4.2 and earlier, `pause_minority` with an odd number of nodes is still the safe choice.

What should `vm_memory_high_watermark` be in production?

Leave it at the 0.6 default unless you have measured a reason to change it. The RabbitMQ docs recommend 0.4 to 0.7, want at least 30% of memory left to the OS, and say values above 0.7 need solid memory monitoring. Nodes running quorum queues need extra headroom for the write-ahead log.

How many nodes should a production RabbitMQ cluster have?

Three, or five if you need to survive two node failures. Use an odd number: a four-node cluster tolerates only one failure, the same as three, and a two-and-two split leaves neither side with a majority.

Can I back up RabbitMQ with VM snapshots?

Not reliably. The RabbitMQ backup guide discourages copying message data from a running node and says a restore only works on a node with exactly the same name. Export definitions on a schedule instead, and use a standby cluster, federation or shovel for message-level disaster recovery.

Is a single RabbitMQ node enough for production?

Only if you accept it as a single point of failure. One node has no replicas to fall back on, so a host failure or a maintenance restart stops the service. Production environments that must survive a node failure run three nodes with quorum queues. A single node is reasonable for development and for workloads you can rebuild.

How much CPU and memory does a RabbitMQ node need?

The RabbitMQ production guidelines give a minimum of 4 CPU cores and 4 GiB of RAM per node and warn against running on a single core. Real sizing depends on message rates, queue count and how long messages sit in queues, so treat the minimum as a floor.

Should I use classic queues or quorum queues in production?

Use quorum queues for anything that must survive a node failure. Classic queues have not been mirrored since RabbitMQ 4.0, so they are single-replica and deliver from their leader only; they still suit transient or rebuildable workloads.

Related