

RabbitMQ Alert Rules Pack: 12 Prometheus Alert Rules to Monitor RabbitMQ Queues and Nodes

These are twelve Prometheus alerting rules for RabbitMQ that catch real incidents, ready to paste into a rules file, with the threshold to adapt and the reason each one fires. Every metric name was checked against the RabbitMQ 4.4 Prometheus documentation and the official RabbitMQ Grafana dashboards on 29 October 2026. For what to monitor and why, read the monitoring and alerting guide; this page is the rules.



By **Scott Sternloff**, Senior Enterprise Architect

LinkedIn • Updated October 29, 2026 • 10 min read • 9 sections

— THE RULE WORTH HAVING FIRST

Alert on a queue that has messages and no consumers. It is the earliest reliable signal that something has broken, it fires before depth looks unusual, and it is never normal. The mistake to avoid is alerting on queue depth alone: it is noisy enough that people mute it, and then it is not there for the incident it was meant to catch. Depth rising while consumers sit at zero is the signal; depth on its own is a graph.

RULE	FIRES WHEN	ACTION
RabbitMQTargetDown	A node stops answering scrapes	Page
RabbitMQMemoryAlarm	Resident memory at the watermark	Page
RabbitMQDiskAlarm	Free disk at the limit	Page
RabbitMQQueueNoConsumers	Messages waiting, zero consumers	Page
RabbitMQQueueGrowing	Depth above your ceiling and still rising	Page
RabbitMQQuorumCritical	A node cannot be lost without a quorum queue losing its majority	Page
RabbitMQMemoryNearWatermark	Memory above 90% of the watermark	Ticket
RabbitMQDiskNearLimit	Free disk under twice the limit	Ticket
RabbitMQUnackedStuck	Unacked high and not moving	Ticket
RabbitMQFileDescriptorsHigh	Open files above 80% of the limit	Ticket

RULE	FIRES WHEN	ACTION
RabbitMQConnectionChurn	Connections opening faster than your baseline	Ticket
RabbitMQUnroutableMessages	Messages dropped as unroutable	Ticket

How to use this pack

Paste the scrape jobs below into your Prometheus configuration, paste the three rule groups into one rules file, then replace every value marked **# ADAPT** with a number from your own baseline. A queue that normally holds 40,000 messages needs a different ceiling from one that normally holds four, and a pasted-in default is how alerts end up muted. Severity **page** means an alert requiring immediate action, so wake someone; **ticket** means look in working hours.

The monitoring and alerting guide explains what each signal means and how to choose thresholds; this page assumes you have read it, or do not need to. Metric names change between plugin versions, so if a rule never evaluates, check the name against your own **/metrics** output before anything else.

Prometheus scrape config: two jobs, because per-queue alerts need per-queue metrics

Scrape **/metrics** for node-level rules and **/metrics/detailed** for per-queue rules. The Prometheus plugin returns aggregated metrics on **/metrics** by default, and the RabbitMQ docs say per-object alerting is not possible with aggregation. The detailed endpoint returns only the metric families you ask for, with the prefix **rabbitmq_detailed_**, and the docs report it can be up to 60 times more efficient

than `/metrics/per-object` for this use. The docs recommend a scrape interval of at least 15 seconds and a rate window of four or more scrape intervals.

```
scrape_configs:
- job_name: rabbitmq                                # node-level metrics, prefix rabbitmq_
  scrape_interval: 15s
  metrics_path: /metrics
  static_configs:
    - targets: ["rmq-1:15692", "rmq-2:15692", "rmq-3:15692"] # ADAPT

- job_name: rabbitmq-queues                          # per-queue metrics, prefix rabbitmq_detail
  scrape_interval: 30s
  metrics_path: /metrics/detailed
  params:
    family: [queue_coarse_metrics, queue_consumer_count]
    # vhost: [production] # optional: limit to one or more vhosts
  static_configs:
    - targets: ["rmq-1:15692", "rmq-2:15692", "rmq-3:15692"] # ADAPT
```

Node alert rules: down, memory, disk, file descriptors, churn

These seven rules use node metrics from `/metrics`. The memory rules compare resident memory with the memory high watermark the node reports, which tracks the alarm closely; for the exact alarm state, use the health check endpoint in section 5.

```
groups:
- name: rabbitmq-node
  rules:
    - alert: RabbitMQTargetDown
      expr: up{job="rabbitmq"} == 0
      for: 1m
      labels: { severity: page }
      annotations:
        summary: "RabbitMQ node {{ $labels.instance }} is not answering scrapes"

    - alert: RabbitMQMemoryNearWatermark
      expr: rabbitmq_process_resident_memory_bytes / rabbitmq_resident_memory_li
```

```

for: 5m
labels: { severity: ticket }
annotations:
  summary: "{{ $labels.instance }}" is above 90% of its memory watermark"

- alert: RabbitMQMemoryAlarm
  expr: rabbitmq_process_resident_memory_bytes >= rabbitmq_resident_memory_l
  for: 0m
  labels: { severity: page }
  annotations:
    summary: "{{ $labels.instance }}" has reached its memory watermark: publi

- alert: RabbitMQDiskNearLimit
  expr: rabbitmq_disk_space_available_bytes < 2 * rabbitmq_disk_space_availa
  for: 10m
  labels: { severity: ticket }
  annotations:
    summary: "{{ $labels.instance }}" has less than twice its free disk limit

- alert: RabbitMQDiskAlarm
  expr: rabbitmq_disk_space_available_bytes <= rabbitmq_disk_space_available
  for: 0m
  labels: { severity: page }
  annotations:
    summary: "{{ $labels.instance }}" is at its free disk limit: publishers a

- alert: RabbitMQFileDescriptorsHigh
  expr: rabbitmq_process_open_fds / rabbitmq_process_max_fds > 0.8
  for: 10m
  labels: { severity: ticket }
  annotations:
    summary: "{{ $labels.instance }}" is using over 80% of its file descripto

- alert: RabbitMQConnectionChurn
  expr: rate(rabbitmq_connections_opened_total[5m]) > 10 # ADAPT: set from
  for: 15m
  labels: { severity: ticket }
  annotations:
    summary: "{{ $labels.instance }}" is opening connections faster than its

```

- **TargetDown, page after 1 minute.** One node down on a three-node cluster is survivable and still worth waking someone, because losing the second one is not. `up` is the series Prometheus itself records for every target.

- **MemoryNearWatermark, ticket.** Publishers block the moment the alarm trips, so the useful alert is the approach. Repeated approaches are a sizing finding, not an operational one.
- **MemoryAlarm and DiskAlarm, page immediately.** When either fires, publishing on that node has already stopped. There is no value in a delay.
- **DiskNearLimit, ticket.** Quorum queues and streams keep everything on disk, so a backlog costs disk, not memory, and the disk limit is reached faster than people expect.
- **FileDescriptorsHigh, ticket.** Descriptor exhaustion stops new connections. Connections, queues and quorum queue segment files all count against the limit, which the docs say should be at least 50,000.
- **ConnectionChurn, ticket.** An application opening a connection per operation instead of reusing one shows up as broker CPU and descriptor use long before anyone links it to the client. The threshold is yours: measure a normal week first.

Queue alert rules: no consumers, growing, stuck

These three rules use the per-queue metrics from the `rabbitmq-queues` job, labelled by vhost and queue. On a cluster with thousands of queues, filter the job by vhost or queue name rather than alerting on every queue.

```
- name: rabbitmq-queues
  rules:
    - alert: RabbitMQQueueNoConsumers
      expr: rabbitmq_detailed_queue_consumers == 0 and rabbitmq_detailed_queue_m
      for: 2m
      labels: { severity: page }
      annotations:
        summary: "{{ $labels.vhost }}/{{ $labels.queue }}" has messages and no co

    - alert: RabbitMQQueueGrowing
      expr: |
        delta(rabbitmq_detailed_queue_messages[15m]) > 0
```

```

        and rabbitmq_detailed_queue_messages > 10000      # ADAPT: your normal c
for: 15m
labels: { severity: page }
annotations:
    summary: "{{ $labels.vhost }}/{{ $labels.queue }}" is above its ceiling a

- alert: RabbitMQUnackedStuck
  expr: |
    rabbitmq_detailed_queue_messages_unacked > 100      # ADAPT: above normal
    and changes(rabbitmq_detailed_queue_messages_unacked[10m]) == 0
for: 10m
labels: { severity: ticket }
annotations:
    summary: "{{ $labels.vhost }}/{{ $labels.queue }}" has unacked messages t

```

- **QueueNoConsumers, page after 2 minutes.** A queue with messages and nothing reading them is never normal, and this fires before depth has grown enough to be obvious. Exclude queues that are drained on a schedule by adding a label filter.
- **QueueGrowing, page after 15 minutes.** Depth alone is noisy; depth above your ceiling that only goes up for fifteen minutes is not. Set the ceiling per queue or per vhost from your own baseline, not a round number.
- **UnackedStuck, ticket.** Consumers are holding messages they never acknowledge, usually blocked on a downstream call with no timeout. It catches a stall that depth misses, because the messages have technically been delivered.

Quorum and routing alert rules

The old version of this pack alerted on a quorum member count metric; we could not find one in the current documentation or the official dashboards, so this rule now probes the management API health check instead. It needs the Prometheus blackbox exporter with an HTTP module that sends credentials for a monitoring user: treat it as a rule to adapt, not to paste.

```

- name: rabbitmq-quorum-and-routing
  rules:
    - alert: RabbitMQQuorumCritical
      # ADAPT: requires blackbox_exporter probing, on every node,
      #   http://:15672/api/health/checks/node-is-quorum-critical
      expr: probe_success{job="rabbitmq-quorum-critical"} == 0
      for: 5m
      labels: { severity: page }
      annotations:
        summary: "{{ $labels.instance }}" cannot be stopped without a quorum queue

    - alert: RabbitMQUnroutableMessages
      expr: rate(rabbitmq_global_messages_unroutable_dropped_total[5m]) > 0
      for: 10m
      labels: { severity: ticket }
      annotations:
        summary: "{{ $labels.instance }}" is dropping unroutable messages"

```

- **QuorumCritical, page after 5 minutes.** The endpoint returns 503 when there are quorum queues that would lose their majority if that node stopped. On a three-replica queue that means a member is already down: the queue still works, which is why nobody notices until the next failure. It is also the check to run before any maintenance.
- **UnroutableMessages, ticket.** A publish that matches no binding is discarded by default unless the exchange has an alternate exchange, so this counter is often the only sign that messages are being lost. Add an alternate exchange and the problem becomes visible in a queue instead.

The load balancer health check most setups get wrong

Point the load balancer at an HTTP health check endpoint, not a TCP check against 5672. A node in a memory or disk alarm still accepts TCP connections and fails every publish, so a TCP check reports it healthy and the balancer keeps sending traffic to a broker that cannot take it.

The management API has endpoints that answer 200 or 503:

`/api/health/checks/local-alarms` fails when the node has an alarm in effect, and `/api/health/checks/ready-to-serve-clients` combines in-service, an AMQP listener being up and the node being below its connection limit. The CLI equivalent is below, but the monitoring guide notes that every CLI call joins Erlang distribution, so it is too heavy to run as a frequent probe.

```
# HTTP, for load balancer probes (needs a monitoring user)
curl -s -o /dev/null -w "%{http_code}" -u monitor:*** \
  http://rmq-1:15672/api/health/checks/local-alarms

# CLI, for scripts and one-off checks
rabbitmq-diagnostics -q check_running && rabbitmq-diagnostics -q check_local_alarm
```

Kubernetes readiness probes are a different case. The RabbitMQ docs say the Cluster Operator's TCP check on the AMQP port, with no liveness probe, is the best practice there, because a probe that waits for a fully rejoined node can block a rolling restart. Use the HTTP checks for client traffic routing, not for pod readiness.

Seven Grafana dashboard panels to build alongside the rules

Alerts tell you something is wrong; these seven panels tell you what. The official RabbitMQ Grafana dashboards cover most of them, and the metric for each is listed so you can build it into an existing board.

PANEL	METRIC	WHAT IT TELLS YOU
Messages ready and unacked, per queue	<code>rabbitmq_detailed_queue_messages_ready</code> , <code>rabbitmq_detailed_queue_messages_unacked</code>	Whether the problem is throughput or acknowledgement
Consumers per queue	<code>rabbitmq_detailed_queue_consumers</code>	The first thing to check in almost every incident
Publish and deliver rates	<code>rabbitmq_global_messages_received_total</code> , <code>rabbitmq_global_messages_delivered_total</code>	A gap between them is the backlog formed, visible before depth shows
Memory and disk headroom per node, watermark drawn	<code>rabbitmq_process_resident_memory_bytes</code> against <code>rabbitmq_resident_memory_limit_bytes</code> ; disk against <code>rabbitmq_disk_space_available_limit_bytes</code>	The watermark line is what marks the graph actionable.
Quorum queue leader elections	<code>rabbitmq_raft_term</code> (Raft dashboard)	A term that keeps rising means repeated elections, an unstable role or network.
Connections and channels per node	<code>rabbitmq_connections</code> , <code>rabbitmq_channels</code> , <code>rabbitmq_connections_opened_total</code>	Churn and leaks both show here first.
File descriptors against the limit	<code>rabbitmq_process_open_fds</code> against <code>rabbitmq_process_max_fds</code>	Invisible until connections start failing.

Table • Dashboard panels and the metrics behind them. Scroll sideways on small screens.

Generate notifications: load the rules and route them with Alertmanager

The rules only evaluate; a firing rule does not trigger a notification until Alertmanager is in the path. Prometheus sends firing alerts to Alertmanager, which groups them, removes duplicates and sends the notification to Slack, email or a paging service. Three steps get the pack from a file to a phone.

1. **Enable the RabbitMQ Prometheus plugin.** `rabbitmq-plugins enable rabbitmq_prometheus` on every node. It serves metrics on port 15692 by default, which is the port the scrape jobs in section 1 use.
2. **Configure Prometheus to load the rules and find Alertmanager.** Add the rules file under `rule_files` and point `alerting` at your Alertmanager, then check the file with `promtool check rules` before reloading.
3. **Route by the severity label.** The pack sets `severity` to `page` or `ticket`, and a child route sends each to a different receiver. If your Alertmanager setup already uses `critical` and `warning`, `page` maps to critical and `ticket` to warning; rename the labels in the rules rather than adding a translation layer.

```
# prometheus.yml
rule_files:
  - /etc/prometheus/rules/rabbitmq.yml          # ADAPT: where you saved the pack
alerting:
  alertmanagers:
    - static_configs:
      - targets: ["alertmanager:9093"]          # ADAPT

# alertmanager.yml
route:
  receiver: ticket-queue
  group_by: [alertname, instance]
  routes:
    - receiver: on-call
      matchers:
        - severity="page"
receivers:
  - name: on-call          # ADAPT: Slack channel, PagerDuty or similar
  - name: ticket-queue     # ADAPT: email or ticketing webhook
```

Alertmanager's defaults wait 30 seconds before the first notification for a group and repeat an unresolved one every 4 hours. Those suit the ticket rules; for the page rules, consider a shorter repeat interval so an unacknowledged alert is not quiet for four hours. Group by `instance` so a node that trips the memory, disk and descriptor rules sends one notification, not three.

Getting the rules running on your cluster

Most teams get these rules running in an afternoon; the slow part is setting the adapt values from a baseline. If you would rather have the rules, dashboards and on-call routing built and tuned for you, monitoring build-out is part of an AceMQ health check, and under a support contract our engineers take the page when a rule fires. We reply within 1 hour to a request.

Frequently asked questions

Which Prometheus metrics should I alert on for RabbitMQ?

Node availability (up), resident memory against `rabbitmq_resident_memory_limit_bytes`, free disk against `rabbitmq_disk_space_available_limit_bytes`, open file descriptors against the limit, connection open rate, and per queue: consumers, total messages and unacknowledged messages from the `rabbitmq_detailed_queue_*` metrics. Add a quorum-critical health check and the unroutable-dropped counter.

Why do my per-queue RabbitMQ alerts never fire?

Because the Prometheus plugin aggregates metrics on `/metrics` by default, so there are no per-queue series to alert on. Scrape `/metrics/detailed` with the `queue_coarse_metrics` and `queue_consumer_count` families, or `/metrics/per-object`, and note that the detailed endpoint uses the `rabbitmq_detailed_` prefix.

Why does my load balancer think RabbitMQ is healthy when it is not?

A TCP check against port 5672 only proves the port is open. A node in a memory or disk alarm still accepts connections and fails every publish. Probe the management API's `/api/health/checks/local-alarms` or `/api/health/checks/ready-to-serve-clients` endpoints instead; they return 503 when the node cannot do work.

How often should Prometheus scrape RabbitMQ?

The RabbitMQ Prometheus guide recommends a scrape interval of at least 15 seconds in production, with RabbitMQ's `collect_statistics_interval` at 10 seconds, and a `rate()` window of at least four scrape intervals.

Related