# VMware Tanzu RabbitMQ

## For Geographically Distributed Systems

## Executive Summary

Modern enterprise applications increasingly span multiple data centers or cloud regions, demanding a messaging backbone that can maintain consistency, reliability, and performance across geographically distributed environments. RabbitMQ clustering across multiple regions provides high availability, scalability, and operational resilience.

This document explains how to design, deploy, and configure a resilient Tanzu RabbitMQ cluster for a geographically distributed system and benefit of using Khepri and Quorum queue features as part of the product. It covers essential aspects such as cluster formation requirements, network and port design, installation and join procedures, and high-availability load-balancing patterns using HAProxy.

It also includes guidance for monitoring and centralized logging using Prometheus, Grafana, and the ELK stack, along with disaster recovery and troubleshooting recommendations to ensure business continuity and operational stability in distributed environments.

Through this approach, enterprises can deploy a highly available, consistent, and fault-tolerant messaging platform optimized for distributed workloads and modern cloud infrastructures.

## Introduction

Deploying RabbitMQ clusters across multiple regions provides several key advantages:

- Improved performance for local clients: Clients can connect to their nearest regional RabbitMQ node, reducing network latency and improving message throughput.
- Enhanced fault tolerance: If one region or zone becomes unavailable, local clients in other regions continue operating without service interruption.
- Optimized resource utilization: Workloads can be distributed regionally, preventing congestion and improving overall responsiveness.

The Khepri database further enhances these benefits by maintaining consistent metadata and cluster configuration across all regions. It enables deterministic leader elections, rapid recovery from failures, and strong data consistency — crucial for distributed deployments where network conditions vary.

When combined with quorum queues, which replicate messages across multiple nodes, Tanzu RabbitMQ ensures durability, data integrity, and continuous availability even during node or regional outages.

In enterprise deployments, Tanzu RabbitMQ provides the same powerful clustering and data consistency capabilities with the added advantage of enterprise-level technical support. This ensures a stable, secure, and fully supported messaging platform for mission-critical, geo-distributed systems.

This document provides a comprehensive guide to deploying Tanzu RabbitMQ clusters on AWS, covering cluster formation, peer discovery configuration, network requirements, quorum queue setup, and operational best practices. It also includes monitoring, disaster recovery, and troubleshooting guidance to help organizations maintain consistent performance and reliability in distributed environments.

# Goals

The primary goals of this document are to define and guide the deployment of a resilient, high-performance Tanzu RabbitMQ cluster that operates reliably across geographically distributed regions.

**Enable Geo-Resilient Clustering:**
Deploy a RabbitMQ cluster with nodes distributed across multiple data centers or cloud regions, ensuring high availability and regional fault tolerance.

**Optimize Performance for Local Clients:**
Improve message delivery speed and throughput by allowing clients to connect to their nearest regional RabbitMQ node, minimizing latency and enhancing responsiveness.

**Ensure Consistent Metadata and State Management:**
Utilize the Khepri database to maintain consistent cluster metadata, user configurations, and policies across distributed nodes, even during network partitions or leader transitions.

**Guarantee Message Durability and Consistency:**
Configure quorum queues to replicate messages across multiple regions, preventing data loss and ensuring consistent delivery across zones.

**Simplify Deployment and Lifecycle Management:**
Leverage the Tanzu RabbitMQ RPM package for standardized installation, upgrades, and configuration across distributed environments.

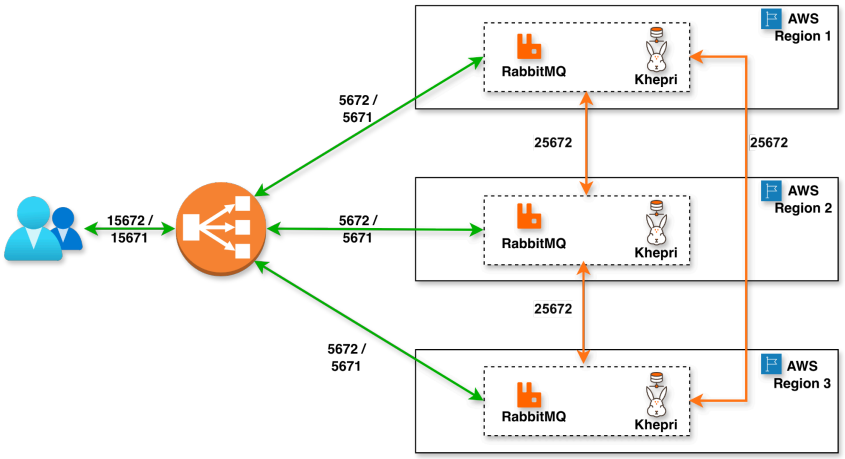
**Implement Comprehensive Monitoring and Recovery:**
Establish observability using Prometheus and Grafana, and define recovery procedures for node, region, or network failures to ensure operational continuity.

## High-level architecture

- Three geographic regions (Region1, Region2, Region3).
- One RabbitMQ node in each Region: rabbit@node1, rabbit@node2, rabbit@node3.
- Global Load Balancer (GLB) / DNS that routes clients to the nearest regional load balancer.
- Regional ingress load balancer or HAProxy that balances client connections to local node(s).
- Khepri metadata store for Raft-based consistency (recommended for Tanzu RabbitMQ features).
- Quorum queues used for critical queues.

## RabbitMQ Cluster Architecture

The diagram below illustrates a 3-node RabbitMQ cluster with Load Balancer and clients.



# Pre-requisites and design decisions

**Minimum nodes:** 3 (recommended), distributed across separate fault domains or different geographical locations (e.g., availability zones or regions).**Time sync:** NTP on all nodes; clock skew causes issues with certificates and leader elections.

**DNS / hostnames:** Each node should have a stable hostname/FQDN that resolves to a reachable IP from other nodes.

**Erlang cookie:** A single identical Erlang cookie across all nodes (/var/lib/rabbitmq/.erlang.cookie) with permissions 400 and ownership rabbitmq:rabbitmq.

**Network performance:** Measure inter-region latency; clustering requires consistent, reliable network connectivity across nodes with low latency, as high latency degrades inter-node communication and replication.

**Storage:** Use local persistent storage with appropriate IO or cloud block storage with high IOPS. Quorum queues replicate across nodes, so disk IO matters.

**Security:** Plan TLS for client connections (5671) and consider TLS for inter-node connections if your network traverses untrusted links.

**Monitoring & logging:** Deploy Prometheus + Grafana, enable RabbitMQ exporter, and centralize logs, which is supported by ELK stack.

# Key ports (Networking)

Make sure these ports are reachable between nodes and from load balancers/clients:

- **AMQP (client):** 5672 (TCP) — client connections (also 5671 for AMQPS/TLS)
- **Erlang distribution / inter-node:** 25672 (TCP) — node-to-node Erlang distribution
- **EPMD:** 4369 (TCP) — Erlang Port Mapper Daemon (node discovery)
- **Management UI (optional):** 15672 (TCP)
- **CLI / automation (HTTP API):** 15672 (HTTP)
- Other: if you enable STOMP/MQTT or streaming, open corresponding ports.

**Note:** Ensure that any firewall, cloud security group, or network ACL permits node-to-node traffic on 25672 and 4369. If using NAT, consistent routing or DNS that resolves to reachable IPs is required. Refer https://www.rabbitmq.com/docs/networking#ports for the entire list of ports used by RabbitMQ.

## Cluster Formation Requirements

- All nodes in the cluster must run compatible versions of Tanzu RabbitMQ and Erlang.
- Ensure consistent, reliable network connectivity across nodes with low latency.
- Configure hostnames or Fully Qualified Domain Names (FQDNs) for all nodes.
- Open necessary ports (25672 for inter-node communication, 4369 for epmd, 5672 for AMQP, 15672 for management UI).
- Use identical Erlang cookies across nodes for authentication.
- Allocate sufficient disk space, memory, and CPU on all nodes. Set resource reservations (CPU/memory) for nodes.
- Deploy Prometheus and Grafana for observability.

## Detailed Implementation Steps

The cluster was provisioned on AWS using Tanzu RabbitMQ RPM packages for installation and lifecycle management. Each RabbitMQ node resides in a separate AWS region, connected via secure private links and governed by dedicated security groups allowing inter-node communication ports (4369, 25672).

This topology ensures realistic testing of geo-resilient RabbitMQ clustering with region-specific clients accessing local nodes through regional load balancers.

**Note:** This setup was prepared using the **Tanzu RabbitMQ RPM package**, however, the overall process and configuration steps remain the same for other Tanzu RabbitMQ installation methods such as tar, OVA or container-based deployments.

## AWS Infrastructure Configuration

For this deployment, a three-node Tanzu RabbitMQ cluster was distributed across three AWS regions to validate cross-region clustering and resilience:

| | AWS Location | Node Hostname |
|---|---|---|
| us-east-2 | Ohio (USA) | rabbit@node1 |
| ap-south-2 | Hyderabad (India) | rabbit@node2 |
| eu-central-1 | Frankfurt (Germany) | rabbit@node3 |

This topology ensures that each node resides in a distinct geographic region, providing resilience against localized infrastructure failures and enabling low-latency access for clients within their respective regions.

## Network and Security Group Configuration

Because RabbitMQ clustering depends on direct inter-node communication, the security groups and firewall rules must allow bidirectional traffic between all three nodes across the following ports:

| Port | Protocol | Description | Required Between |
|---|---|---|---|
| **4369** | TCP | Erlang Port Mapper Daemon (EPMD) | All cluster nodes |

| | | | |
|---|---|---|---|
| **25672** | TCP | Erlang inter-node communication | All cluster nodes |
| **5672** | TCP | AMQP client connections | Load balancer, clients → nodes |
| **5671** | TCP | AMQPS (for TLS) client connections | Load balancer, clients → nodes |
| **15672** | TCP | RabbitMQ Management UI / HTTP API | Admin clients → nodes |
| **1883 / 8883** | TCP | MQTT (optional) | Clients → nodes |
| **15675** | TCP | Web STOMP (optional) | Clients → nodes |

## AWS Security Group Setup

1. **Create a security group per region:**

   For example:

   - sg-rabbitmq-us-east-2
   - Sg-rabbitmq-ap-south-2
   - sg-rabbitmq-eu-central-1

2. **Inbound rules (per security group):**

   - Allow inbound TCP traffic on the ports listed above

- For clustering (ports 4369 and 25672), restrict access to the private IPs or security group IDs of the other two RabbitMQ nodes
- For client access (ports 5672, 5671, 15672), allow inbound traffic only from:
  - The corresponding regional load balancer
  - Approved application subnets or VPN CIDRs.

3. **Outbound rules:**

Allow all outbound traffic (default AWS SG behavior), or restrict to specific peer nodes if you enforce tight egress policies.

4. **Verify DNS or private interconnectivity:**
Ensure that all nodes can resolve and reach each other via their private DNS names or Elastic IPs.
Example:

```
ping node1.example.internal

ping node2.example.internal

ping node3.example.internal
```

## Key Recommendations

- Use private inter-region VPC peering or AWS Transit Gateway to enable secure internal communication across regions.
- Maintain consistent hostnames (e.g., via Route 53 private DNS) for all nodes to ensure stable Erlang cluster formation.
- Always test latency and connectivity using telnet or nc to confirm port reachability across nodes.
- If using TLS for inter-node communication, ensure that certificates are trusted across all regions and nodes.

# RabbitMQ Node Configuration (Within the AWS EC2 Instances)

- Prepare all nodes (at least 3 nodes) with RHEL or CentOS and ensure network connectivity.
- Install dependencies.

```shell
sudo dnf update -y

sudo dnf install -y erlang
```

- Install Tanzu RabbitMQ RPM.

```shell
sudo rpm -ivh
tanzu-rabbitmq-server-<version>-1.el8.x
86_64.rpm
```

- Configure Erlang cookie and copy it to others.

On the primary node (node1) create a cookie and copy it to others:

```shell
echo 'SECRET_COOKIE' >
/var/lib/rabbitmq/.erlang.cookie

# copy to node2 and node3 (use scp)

scp /var/lib/rabbitmq/.erlang.cookie
root@node1:/tmp/
```

```
ssh root@node1 "sudo mv
/tmp/.erlang.cookie
/var/lib/rabbitmq/.erlang.cookie &&
sudo chown rabbitmq:rabbitmq
/var/lib/rabbitmq/.erlang.cookie &&
sudo chmod 400
/var/lib/rabbitmq/.erlang.cookie"

# repeat for node3
```

- Enable and start the RabbitMQ service.

Shell

```
sudo systemctl enable
tanzu-rabbitmq-server

sudo systemctl start
tanzu-rabbitmq-server

sudo systemctl status
tanzu-rabbitmq-server
```

- Enable management plugin (optional)

Shell

```
sudo rabbitmq-plugins enable
rabbitmq_management

# reload or restart if required
```

```
sudo systemctl restart
tanzu-rabbitmq-server
```

- Create admin user & enable TLS (recommended)

```shell
# create admin user and set
permissions

sudo rabbitmqctl add_user admin
StrongPassword

sudo rabbitmqctl set_user_tags admin
administrator

sudo rabbitmqctl set_permissions -p /
admin ".*" ".*" ".*"
```

For TLS, upload certs and configure /etc/rabbitmq/rabbitmq.conf for TLS listeners.

## Cluster formation — join nodes

**Option A:** Manual Cluster Formation (when explicit control of join order is required.)

On node1 (first node), start normally. Add additional nodes to the cluster.

- Stop RabbitMQ service on the joining node (node2 and node3).

```shell
# on node2
```

```
sudo systemctl stop
tanzu-rabbitmq-server

sudo rabbitmqctl stop_app

sudo rabbitmqctl reset

sudo rabbitmqctl join_cluster
rabbit@node1

sudo rabbitmqctl start_app


# verify

sudo rabbitmqctl cluster_status
```

Repeat for node3. The canonical safe sequence is:

- stop_app, reset (if necessary), join_cluster, start_app.

**Option B:** Automatic Cluster Formation with Config File Peer Discovery (classic_config)

- Enable the plugin:

Shell

```
sudo rabbitmq-plugins enable
rabbitmq_peer_discovery_config
```

- Configure /etc/rabbitmq/rabbitmq.conf on all nodes:

```
None
cluster_formation.peer_discovery_backend = classic_config

cluster_formation.classic_config.nodes.1 = rabbit@node1

cluster_formation.classic_config.nodes.2 = rabbit@node2

cluster_formation.classic_config.nodes.3 = rabbit@node3
```

- Restart RabbitMQ:

```
Shell
sudo systemctl restart tanzu-rabbitmq-server
```

Nodes will discover peers and form a cluster automatically at startup.

- Validate cluster

```
Shell
sudo rabbitmqctl cluster_status

# Example output shows the three nodes under running_nodes

sudo rabbitmq-diagnostics ping
```

```
sudo rabbitmqctl report   # collects diagnostics
```

Firewall and SELinux

- Open ports with firewall-cmd (example):

```shell
sudo firewall-cmd --permanent --add-port=5672/tcp

sudo firewall-cmd --permanent --add-port=25672/tcp

sudo firewall-cmd --permanent --add-port=4369/tcp

sudo firewall-cmd --permanent --add-port=15672/tcp

sudo firewall-cmd --reload
```

- If SELinux is enforced, ensure RabbitMQ file contexts and ports are allowed or configure appropriate booleans/contexts per vendor docs.

## Monitoring and Troubleshooting

**Prometheus:**

Use Prometheus and the RabbitMQ exporter to scrape metrics; create Grafana dashboards (queues, consumers, message rates, unacked).

Monitor these key signals:

- rabbitmq_queue_messages_ready
- rabbitmq_queue_messages_unacknowledged
- erlang_memory_used
- rabbitmq_node_is_running
- Node partition events / leader election events

**Centralized Logging:**

For centralized log management, implement an ELK stack (Elasticsearch, Logstash, and Kibana) or its cloud equivalent.

- **Elasticsearch** stores and indexes RabbitMQ logs from all nodes.
- **Logstash** collects and parses log data from /var/log/rabbitmq/ or systemd journal.
- **Kibana** provides a visual dashboard to analyze trends, errors, and operational events across regions.

This combination of Prometheus/Grafana for metrics and ELK for logs gives complete observability — enabling teams to monitor system health, detect anomalies, and trace message delivery issues across distributed RabbitMQ nodes in real time.

# Validation & Health Checks

- **Cluster membership:** *rabbitmqctl cluster_status*
- **Node health:** *rabbitmq-diagnostics status*
- **Ping:** *rabbitmq-diagnostics --node <node> ping*
- **List queues/consumers:** *rabbitmqctl list_queues name messages consumers*
- **HTTP API:** *curl -u user:pwd http://<RabbitmqServer IP>:15672/api/overview*

## Operational recommendations (Geo Considerations)

- Prefer quorum queues for cross-region deployments (designed for replicated state).
- Avoid synchronous replication of large volumes across regions with high latency — use local queues + federation for async cross-region flow where latency is high.
- Federation vs clustering: For very high latencies consider federation rather than forming a single Erlang cluster. (Erlang clustering across very high latency networks performs poorly.)
- Testing: Simulate failovers, network partitions, and region outages. Measure message delivery, enqueue latency, and time to restore.

## Troubleshooting Common Issues

- Nodes cannot see each other: check 4369 + 25672 connectivity, cookie mismatch, and hostname resolution.
- Cookie errors: ensure /var/lib/rabbitmq/.erlang.cookie content & permissions are identical.
- High memory pressure: RabbitMQ will start throttling publishers; monitor memory metrics and tune vm_memory_high_watermark.
- Slow/dropped connections: examine network performance, per-socket limits, and backlog settings on load balancer.

## Benefits

Deploying a Tanzu RabbitMQ cluster across geographically distributed systems provides several business and technical advantages:

- **High Availability (HA):** With nodes placed in different zones or regions, the cluster can tolerate the failure of a single zone or datacenter without losing service.
- **Resilience & Fault Tolerance:** Quorum queues and Raft-based metadata ensure consistent state replication and prevent data loss during node or region failures.
- **Scalability:** Adding new nodes or regions is straightforward, and load balancers distribute traffic efficiently to handle growing workloads.
- **Performance Optimization:** Clients can be routed to the nearest regional RabbitMQ node, reducing latency for message publishing and consumption.

- **Operational Flexibility:** Using RPMs standardizes installation and upgrades across environments, simplifying patching and compliance.
- **Security Integration:** TLS, consistent Erlang cookies, and centralized access policies enforce secure inter-node and client communications.
- **Monitoring & Observability:** Integration with Prometheus, Grafana, and log collectors provides real-time visibility into health, capacity, and performance across regions.

## Conclusion

By following a structured deployment model, enterprises can operate a resilient, scalable RabbitMQ cluster that spans multiple geographic locations.

Clustering across regions — while challenging due to latency and network partitions — is made manageable by:

- using quorum queues for critical workloads,
- enabling Khepri metadata store for consistency,
- and routing client traffic through global and regional load balancers.

This design allows organizations to deliver low-latency, highly available messaging infrastructure that continues to operate seamlessly even during planned maintenance or unexpected outages.

Ultimately, Tanzu RabbitMQ clustering provides the **foundation for modern, geo-resilient applications**, ensuring reliable message delivery, operational flexibility, and strong business continuity guarantees.

## References


- Tanzu RabbitMQ Documentation: Tanzu RabbitMQ RPM documentation
- RabbitMQ Clustering: Clustering Guide | RabbitMQ
- Distributed RabbitMQ: Distributed RabbitMQ
- RabbitMQ and Erlang compatibility: Erlang Version Requirements | RabbitMQ
- RabbitMQ - Install: Installing RabbitMQ

## A. Kubernetes – Helm (OCI Chart)

Helm is the package manager for Kubernetes. It packages resources into *charts*, making installs and upgrades repeatable.

### Why use it?

- Rapid deployment
- Easy version management
- Rolling upgrades supported by Kubernetes

### TechDocs (Install):

Installation using Helm – VMware Tanzu RabbitMQ on Kubernetes (v4.1)

**Example:**

```
None
helm registry login rabbitmq-helmoci.packages.broadcom.com -u username
-p <your-token>

helm upgrade --install trmq
oci://rabbitmq-helmoci.packages.broadcom.com/tanzu-rabbitmq/tanzu-rabbit
mq --version <VERSION> --namespace messaging --create-namespace -f
values.yaml
```

## B. Kubernetes – Carvel (kctrl / kapp-controller)

Carvel is a suite of tools for app lifecycle management on Kubernetes. `kctrl` is the CLI to install *packages* into clusters.

### Why use it?

- Native to Tanzu Kubernetes Grid (TKG)
- Integrates with Tanzu packaging repository
- Supports multi-cluster consistency

### TechDocs (Install):

Installation using Carvel – VMware Tanzu RabbitMQ on Kubernetes (v4.1)

**Example:**

```
None
kctrl package repository add -r
tanzu-rabbitmq -n
kapp-controller-packaging-global --url
<CARVEL_REPO_URL>

kctrl package install -i trmq -p
tanzu-rabbitmq.<domain> -v <VERSION> -n
messaging --values-file values.yaml
```

## C. OVA (Virtual Appliance)

An OVA is a pre-packaged virtual machine image, ready to run on vSphere or other hypervisors.

### Why use it?

- Turnkey deployment
- Minimal Kubernetes knowledge required
- Includes pre-configured RabbitMQ

### TechDocs (Install):

VMware Tanzu RabbitMQ Virtual Machine (OVA) Installation (v4.1)

Flow: Download → Import into vSphere (or compatible hypervisor) → Power on → Configure via CLI or management UI

## D. RPM (Linux)

RPM is the package format for RHEL, CentOS, and related distros.

### TechDocs (Install):

Installing RabbitMQ on RPM-based Linux

### Why use it?

- Fits traditional VM or bare-metal deployments
- Works with existing OS-level monitoring and automation

### Highlights:

- Ensure Erlang/OTP compatibility.
- Use official RabbitMQ repositories.

Example:

```
None
dnf install -y erlang rabbitmq-server
```

## 2. Dependencies Before Upgrading RabbitMQ Erlang/OTP

- Each RabbitMQ version supports specific Erlang versions — check the compatibility matrix: Erlang Compatibility Matrix.

- Upgrade Erlang first if required.

Example:

```
None
sudo yum install
erlang-<COMPATIBLE_VERSION>
```

Other dependencies

- OpenSSL 1.1+ or 3.x
- Updated `glibc` and `libstdc++`
- Disk space & RAM for metadata store migration
- Kubernetes API version compatibility for K8s installs

## 3. Checking RabbitMQ Status

**Kubernetes:**

```
None
   kubectl -n messaging get pods -l
   app.kubernetes.io/name=rabbitmq
   kubectl exec -it <pod> --
   rabbitmq-diagnostics status
   kubectl exec -it <pod> -- rabbitmqctl
   cluster_status
```

**OVA / RPM:**

```
None
   sudo systemctl status rabbitmq-server
   sudo rabbitmq-diagnostics status
   sudo rabbitmqctl cluster_status
```

**Enable UI:**

```
None
    sudo rabbitmq-plugins enable
    rabbitmq_management
```

Access via: `http://<host>:15672/`

# 4. Pre-Upgrade Checklist

- Read Release Notes — Understand changes in RabbitMQ and Erlang/OTP.
- Verify Erlang/OTP and dependency compatibility
- Confirm cluster health (no alarms, nodes reachable)
- Backup definitions:

```
None
       rabbitmqadmin export
       /tmp/definitions.json
```

- Backup persistent storage (stop and copy data directory)
- Follow supported upgrade paths
- In Kubernetes: verify chart/package version compatibility

# 5. Upgrade Flows

Upgrades in RabbitMQ should ideally be performed using **in-place rolling upgrades**, where nodes are sequentially upgraded while the cluster continues serving traffic.
This method minimizes downtime and disruption, making it the **recommended standard upgrade path** for Tanzu RabbitMQ in most production environments.

**Note:** Rolling upgrades are supported only when the version step is compatible. If the version gap is too large or rolling upgrades are not supported for the target version, a **Blue-Green upgrade strategy** should be followed instead to ensure service continuity.

## A. Kubernetes – Helm

### TechDocs (Upgrade):

Upgrade using Helm – VMware Tanzu RabbitMQ on Kubernetes (v4.1)

```
None
helm upgrade --install trmq
oci://rabbitmq-helmoci.packages.broadcom.com/tanzu-rabbitmq/tanzu-rabbit
mq --version <NEW_VERSION> -n messaging -f values.yaml

kubectl -n messaging rollout status statefulset/trmq
```

## B. Kubernetes – Carvel

### TechDocs (Upgrade):

Upgrade using Carvel – VMware Tanzu RabbitMQ on Kubernetes (v4.1)

```
None
kctrl package installed update -i trmq -n messaging --version
<NEW_VERSION> --values-file values.yaml
```

## C. OVA (Virtual Appliance)

For OVA, **Blue-Green strategy is recommended**..

## TechDocs (Blue-Green):

[Blue-Green Upgrade – RabbitMQ]

### Flow:

1. Deploy **Green OVA** (new version) alongside **Blue OVA** (current).
2. Export definitions from Blue:

```
None
rabbitmqadmin export /tmp/definitions.json
```

3. Import into Green:

```
None
rabbitmqadmin import /tmp/definitions.json
```

4. Set up Federation to sync queues:

```
None
rabbitmqctl set_parameter
federation-upstream blue
'{"uri":"amqp://blue-host"}'

rabbitmqctl set_policy --apply-to queues
migrate-all ".*"
'{"federation-upstream":"blue"}'
```

5. Cut over traffic to Green.
6. Decommission Blue after validation.

## D. RPM

```
None
sudo rabbitmqctl stop
```

```
sudo rpm -Uvh
tanzu-rabbitmq-server-<NEW_VERSION>.<arch
>.rpm
sudo rabbitmqctl start
```

## 6. Blue-Green Upgrade Strategy (General)

- Deploy a parallel **Green** cluster
- Migrate configuration & definitions
- Use Federation/Shovel for queue data migration
- Redirect clients to Green
- Retire Blue once stable

## 7. Khepri Database in RabbitMQ

Starting with RabbitMQ **4.0**, **Khepri** is fully supported but **Mnesia remains the default metadata store** in versions 4.0 and 4.1.

Khepri must be explicitly enabled using the `khepri_db` feature flag.

**Benefits of Khepri:**

- Faster and safer crash recovery
- Handles large metadata volumes efficiently
- More predictable upgrade behavior

Check metadata store in use:

```
None
rabbitmqctl eval
'application:get_env(rabbit,
schema_db_dir).'
```

# 8. Post-Upgrade Validation

```
None
rabbitmqctl status
rabbitmqctl list_feature_flags
rabbitmqctl cluster_status
rabbitmq-diagnostics check_local_alarms
rabbitmq-diagnostics check_port_listener
```

Test:

- Client connections
- Queue throughput
- Management UI access

# 9. Migrating from OSS to VMware Tanzu RabbitMQ

It is often overlooked that the team at Broadcom own, manage and maintain open source RabbitMQ. It is this same team that builds on this to produce the commercial Tanzu RabbitMQ builds. Whether deploying on Kubernetes using the Tanzu RabbitMQ Helm chart or deploying on bare metal using the raw components exclusively available in a tarball. Tanzu RabbitMQ takes the out of the box open source RabbitMQ to the next level and beyond, with robust, production-ready artifacts that empower teams to better manage messaging workloads at scale with greater confidence.

### Why migrate:

Migrating from open-source RabbitMQ to VMware Tanzu RabbitMQ unlocks significant advantages:

- **Enterprise-grade readiness** — Tanzu RabbitMQ includes validated, production-ready images (OVA and OCI) with built-in security, dependency validation, and hardened defaults that minimize operational risk.

- **Enhanced resilience and data safety** — Includes features like Warm Standby Replication for fast failover, automated multi-site replication with declarative config, and intra-cluster compression to optimize network usage.

- **Strong governance and security posture** — Features such as vault integration for credential management, TLS 1.2+ enforced by default, FIPS compliance, and enterprise SLAs ensure secure and reliable operations.

- **Simplified deployment and support** — Tanzu RabbitMQ delivers day-zero-ready deployments with curated images, access to VMware engineering resources, and 24/7 enterprise-grade support — reducing the complexity of installing, upgrading, and securing the platform.

- **Future-proof enhancements** — Native improvements such as AMQP 1.0 support over WebSockets, MQTTv5 for IoT scale, JMS integration, performance optimizations for Quorum Queues, stream filtering, and a revamped metadata store (Khepri) offer ongoing operational and development benefits.

**Steps:**

1. Export definitions from OSS:

```
None
rabbitmqadmin export
/tmp/definitions.json
```

2. Perform the Migration:

   - **Preferred**: Use an in-place rolling upgrade if the version step is supported — upgrade nodes sequentially while the cluster remains online.
   - **Fallback**: If rolling upgrades aren't possible (e.g., due to a major version gap), adopt a Blue-Green migration by running OSS and Tanzu clusters in parallel and syncing traffic/state.

3. Import definitions into VMware Tanzu RabbitMQ:

```
None
rabbitmqadmin import
/tmp/definitions.json
```

4. Switch clients.

5. Decommission OSS RabbitMQ (in-case of blue-green upgrade).

# Conclusion

Upgrading RabbitMQ means checking Erlang/OTP compatibility, backing up, following the right upgrade path, and validating results.

**VMware Tanzu RabbitMQ** offers enhanced stability, tested packaging, enterprise-grade support, and exclusive features like **Warm Standby Replication**, **Intra-Cluster Compression**, **AMQP 1.0 over WebSocket** and **Full Khepri Support** — delivering improved resilience, security, observability, and scalability that go beyond open-source RabbitMQ.

vmware® by Broadcom