CHECKLIST

NoSQL RBAC Production Readiness Checklist

A practical checklist for securing NoSQL databases with role-based access control, including role design, privilege validation, operational separation, and pre-production verification.

NoSQL RBAC Production Readiness Checklist

Use this checklist to review a NoSQL role-based access control design before production. Check each item that applies and document any gaps.

---

1) Define the data and trust boundaries

- [ ] Identify each NoSQL database, namespace, cluster, or environment in scope
- [ ] List the data sets stored in each one
- [ ] Mark sensitive data, regulated data, and operational data separately
- [ ] Identify which services, teams, and humans need access
- [ ] Confirm whether any data paths mix sensitive and non-sensitive records
- [ ] Note any shared database resources that may need extra isolation

2) Map real operational tasks to roles

- [ ] Define roles from tasks, not job titles
- [ ] Create a minimal role for each application function or operational function

- [ ] Separate read, write, and administrative roles
- [ ] Create distinct roles for backup, restore, maintenance, and incident response
- [ ] Avoid reusing one powerful role across unrelated services
- [ ] Make sure each role has a clear owner

3) Minimize privileges

- [ ] Grant only the collections, tables, buckets, or document groups required
- [ ] Remove unused CRUD permissions
- [ ] Restrict list, index, schema, and admin actions unless needed
- [ ] Confirm that no role has broad default access "just in case"
- [ ] Verify that privileged actions are time-bounded when possible
- [ ] Review whether any role can reach more sensitive data than necessary

4) Separate application access from human administration

- [ ] Use different identities for applications and administrators
- [ ] Do not share application credentials between services
- [ ] Do not use shared admin accounts for routine access
- [ ] Ensure on-call or incident roles are distinct from day-to-day access
- [ ] Require elevated access only when a task truly needs it
- [ ] Confirm that human actions can be attributed to an individual identity

5) Validate allowed and denied behavior

- [ ] Test every role against its intended allowed actions
- [ ] Test denied operations explicitly

- [ ] Confirm the application handles authorization failures gracefully
- [ ] Verify that read-only roles cannot write or delete data
- [ ] Verify that service roles cannot access unrelated collections or namespaces
- [ ] Check that administrative actions are blocked for non-admin roles

6) Review application dependencies

- [ ] Inventory all database calls made by the application or service
- [ ] Remove permissions that support legacy behavior no longer needed
- [ ] Check for hidden dependencies in background jobs, retries, and maintenance tasks
- [ ] Confirm that build, test, and staging identities are not overprivileged
- [ ] Verify that schema changes do not silently require broader access
- [ ] Re-test after refactoring or service decomposition

7) Confirm operational controls around elevated access

- [ ] Require approval or process controls for privileged role assignment
- [ ] Limit elevated access duration where supported
- [ ] Log role grants, role changes, and privileged actions
- [ ] Review audit logs for unusual access patterns
- [ ] Ensure break-glass access is documented and controlled
- [ ] Confirm rollback steps for mistaken privilege grants

8) Check vendor and platform specifics

- [ ] Review the exact role and resource model for the NoSQL product in use
- [ ] Confirm how inheritance, scope, and resource matching work
- [ ] Verify whether permissions apply at database, collection, document, or API level

- [ ] Check whether the platform supports fine-grained authorization needs
- [ ] Avoid assuming that similar products grant access the same way
- [ ] Document version-specific behavior that affects security

9) Decide whether RBAC is enough

- [ ] Confirm the access need can be described as a stable operational task
- [ ] Identify any request-based or record-based decisions RBAC cannot express
- [ ] Add application-layer authorization if needed
- [ ] Consider field-level encryption or separate databases for higher-risk data
- [ ] Use data partitioning if trust zones should not share the same access path
- [ ] Validate that the final design limits blast radius if credentials are exposed

10) Pre-production sign-off

- [ ] Roles are minimal and clearly documented
- [ ] Application, human, and admin access are separated
- [ ] Denied-access tests passed
- [ ] Audit logging is enabled and reviewed
- [ ] Privileged access is controlled and traceable
- [ ] Platform-specific behavior has been verified
- [ ] The permission model matches the data model
- [ ] The system is ready for production rollout

Quick review questions

Ask these before approval:

- Can each role be explained in one sentence as a real operational task?
- Would a leaked credential expose only the data and actions it truly needs?
- Are read, write, and administrative privileges separated?
- Have denied operations been tested, not just allowed ones?
- Are there any dynamic authorization needs that RBAC cannot handle alone?
- Would the design still be understandable after a service refactor or team change?

Final note

RBAC is most effective when it is narrow, explicit, and paired with good data design. If a role starts to feel like a catch-all, it is probably time to split it before production.