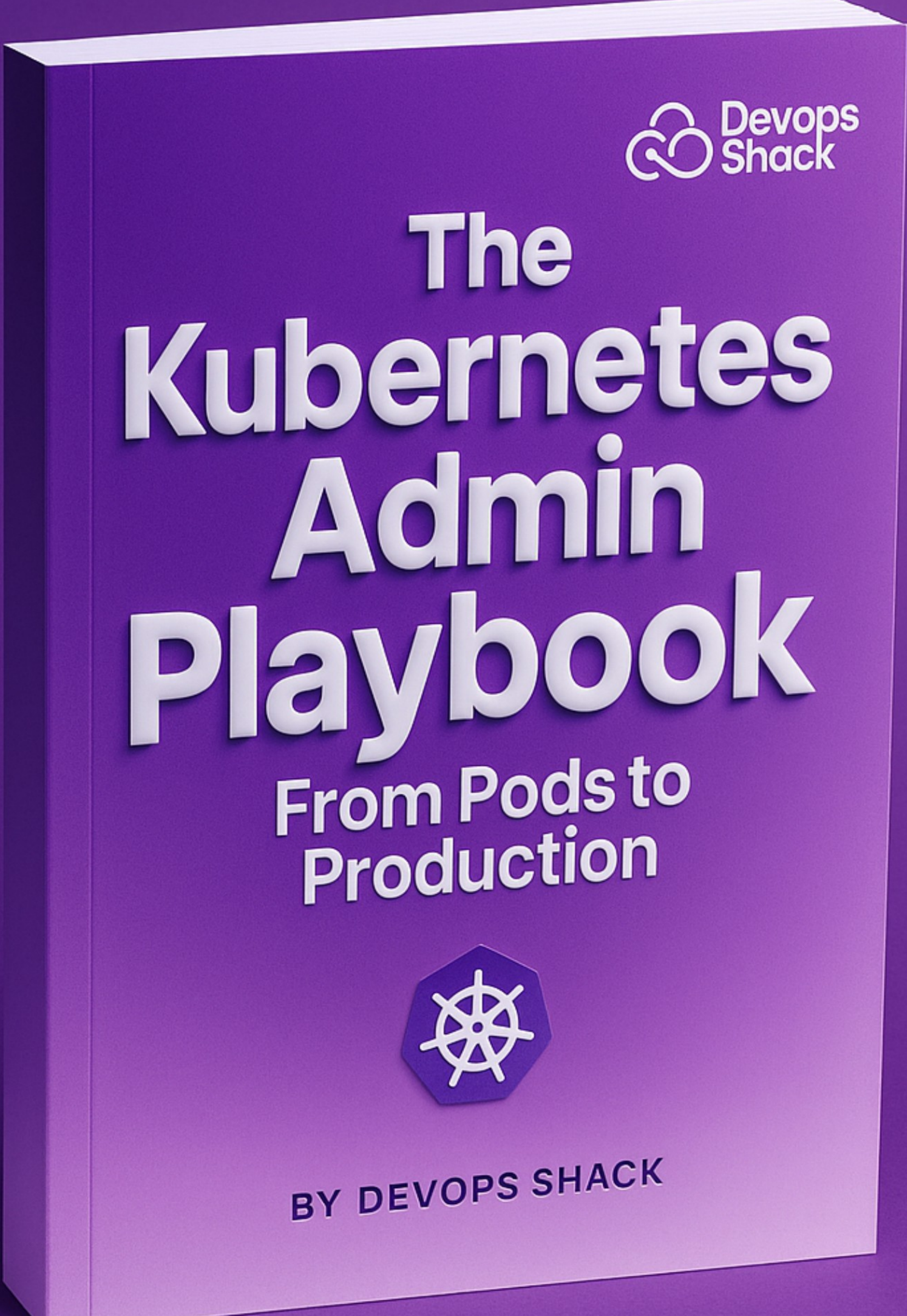

Devops Shack

The Kubernetes Admin Playbook

From Pods to Production

BY DEVOPS SHACK

DevOps Shack

The Kubernetes Admin Playbook: From Pods to Production

Table of Contents

☑ 6. RBAC & Access Control

- ServiceAccounts, Roles, ClusterRoles, who-can
- Least privilege enforcement and namespace isolation

☑ 7. Security Hardening Must-Knows

- Drop capabilities, runAsNonRoot, secrets protection
- NetworkPolicies, admission controllers (OPA/Gatekeeper)

☑ 8. Monitoring, Logging & Observability

- Prometheus + Grafana setup
- Centralized logging with Loki/Fluent Bit
- Key metrics every admin must track

☑ 9. Deployments, Upgrades & Rollbacks

- Rolling updates, surge strategies
- Helm upgrade/rollback, draining nodes, version control

☑ 10. Think Like an SRE: Resilience, Cost, and Chaos

- Designing for failure, cost-efficient scaling
- Chaos testing, disaster recovery plans, self-healing automation

1. From Hello K8s to Real Admin Work

🧠 *Why "kubectl get pods" is just the beginning — not the destination.*

🚀 **Introduction**

You've installed Minikube. You ran kubectl get pods. You deployed a sample app.

It feels good — but here's the brutal truth:

Knowing how to list pods doesn't make you production-ready.

🎯 **What You Think Being a Kubernetes Admin Is:**

- Knowing kubectl commands
- Understanding YAML structure
- Installing Helm charts
- Creating deployments

These are important... but they are Level 1.

💡 **What Being a *Real* Kubernetes Admin Actually Means:**

☑ Understanding **why things break** and how to **recover safely**
☑ Designing systems that **self-heal, scale, and alert you**
☑ Enforcing **RBAC, limits, and security controls**
☑ Ensuring **zero-downtime deployments and rollbacks**
☑ Handling **node failures, storage issues, and outages** under pressure
☑ Building pipelines that deploy **automated, secure, observable, reproducible** infrastructure

🧱 **The Big Mental Shift: From Dev to Infra Operator**

Learner Mode	**Production Admin Mode**
Deploys apps to learn	Deploys apps to serve live users
Focuses on what works	Focuses on what breaks

Learner Mode	Production Admin Mode
Watches metrics after deployment	Sets up alerts *before* deploying
Reads docs	Writes runbooks
Installs tools	Integrates tools into environments
Solves errors by trial	Prevents errors with probes & limits

🔐 Why This Shift Is Critical for Jobseekers

You may not be working in a production team yet, but employers **want to see this mindset** in interviews:

❌ *"I deployed an app on Minikube"*
✅ *"I used Minikube to simulate prod-like failures and built recovery pipelines with liveness probes, HPA, and alerts."*

Same tool.
Different depth.

🎯 What This Document Will Now Do For You

This guide will:

- Take you step-by-step from surface-level knowledge to real admin capabilities
- Cover every major production concern: security, scaling, logging, failures, cost
- Give you **examples**, **commands**, **config tips**, and **debugging techniques**

By the end of this, you'll be able to say in interviews:

"While I haven't managed a live Kubernetes cluster at work yet, I've simulated production setups in my lab, handled crash scenarios, set up probes, enforced limits, configured logging and RBAC, and built self-healing, observable environments."

That's **job-ready talk**.

2. Pod Troubleshooting Deep Dive

If you can't fix a broken pod, you can't survive in production.

🚨 Why This Matters

When something breaks in Kubernetes, it almost always starts with:

❌ *"Pod is in CrashLoopBackOff"*
❌ *"Pod is stuck in Pending"*
❌ *"ImagePullBackOff"*
❌ *"OOMKilled"*

A real Kubernetes Admin doesn't panic.
They debug methodically.

This section will show you exactly how to do that.

🧰 Your Core Toolbox: The 5 Commandments of Pod Debugging

Start with the basics, then go deeper 👇

1. kubectl get pods -n <namespace>

- Check **status** (Pending, Running, CrashLoopBackOff)
- Add -o wide for node info and IPs

✅ Use with:
```
kubectl get pods -n app-space -o wide
```

2. kubectl describe pod <pod-name> -n <namespace>

- Shows detailed event logs, restart counts, scheduling issues
- Best for: CrashLoopBackOff, pending pods, node affinity errors

✅ Tip:
Look for lines like:

```
Last State: Terminated

Reason: OOMKilled

Message: container exited with status 137
```

3. kubectl logs <pod-name> [-c container-name] -n <namespace>

View container logs (STDOUT/STDERR)

- Use -c if there are multiple containers

✅ Pro Tip:
Run kubectl logs <pod> --previous to see logs from the previous failed container.

4. kubectl exec -it <pod> -n <ns> -- sh

- SSH into the pod (if it's running)
- Explore files, environment, DNS resolution

✅ Try:
cat /etc/resolv.conf, env, ping, curl localhost:port

5. kubectl events OR inside kubectl describe

- This is where **you find the root cause** most of the time
- Look for:
    - Image pull failures
    - Pod eviction
    - Node pressure (memory, disk)
    - Scheduling issues

🧠 **Common Pod Failure Scenarios & Fixes**

⚠️ **CrashLoopBackOff**

🔍 Caused by:

- App crashes on start
- Bad readiness probe
- Exception in code

✅ Fix:

Check logs (kubectl logs)

- Review livenessProbe & readinessProbe
- Increase initialDelaySeconds
- Debug with exec to check configs

⚠ **ImagePullBackOff**

🔍 Caused by:

- Typo in image name
- Private image, no pull secret
- DockerHub rate limits

☑ Fix:

- Correct the image tag
- Add pull secrets using imagePullSecrets
- Try pulling image manually from node

⚠ **OOMKilled**

🔍 Caused by:

- App exceeded memory limit

☑ Fix:

- Increase resources.limits.memory
- Optimize app memory usage
- Analyze heap dumps/logs

⚠ **Pending**

🔍 Caused by:

- No matching node (taints, affinity)

PVC not bound

- Node selector mismatch

✅ Fix:

- Check pod spec: tolerations, selectors
- Validate storageClass and PVC
- Use kubectl get events to see scheduler issues

🔎 Bonus Tools for Faster Debugging

Tool	**Purpose**
k9s	Terminal UI to explore pods quickly
kubetail	Tail logs of multiple pods at once
stern	Multi-pod log tailer with filters
lens	GUI-based cluster explorer

📋 Runbook Sample: My Pod Won't Start – What I Do

```
1. kubectl get pods -n myapp
2. kubectl describe pod <name>
3. kubectl logs <name>
4. Check events for pull/image/env issues
5. If it's CrashLoopBackOff:
   - Check readiness probe
   - Check logs for exceptions
   - Use --previous logs if restarting
6. If Pending:
   - Check PVC
   - Check node selector, taints, affinity
```

7. Patch or fix YAML → apply again

Once you can diagnose pods like this, you're already more capable than 80% of Kubernetes users who only "apply YAMLs and hope".

3. Health Checks & Auto-Healing

💡 *A broken pod isn't the problem — the lack of detection and recovery is.*

🧠 **Why Health Checks Matter**

Anyone can deploy a pod.
But a **production admin** ensures:

- The pod doesn't receive traffic when unhealthy ☑
- It restarts automatically when it crashes ☑
- The cluster scales based on real load ☑
- Users never know something broke ☑

This is the essence of **self-healing systems** — the DevOps dream.

🔬 Liveness, Readiness, and Startup Probes

1. Liveness Probe

✸ *"Is my container alive?"*

- If the probe **fails**, Kubernetes will **restart the pod**
- Used to recover from deadlocks or app

crashes
```
livenessProbe:
 httpGet:
  path: /healthz
  port: 8080
 initialDelaySeconds: 10
 periodSeconds: 5
```

2. Readiness Probe

🎯 *"Is my container ready to serve traffic?"*

- If it **fails**, traffic will **not be sent to the pod**
- Prevents users from hitting the app before it's

ready
```
readinessProbe:
 httpGet:
  path: /ready
```

```
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 3
```

3. Startup Probe (since K8s v1.16)

"Is the app slow to start, but still okay?"

- Used for **slow-booting apps** (e.g., Java)
- Temporarily disables liveness & readiness until the app fully

starts

```
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  failureThreshold: 30
  periodSeconds: 5
```

G Testing and Debugging Probes

☑ Use curl localhost:<port>/healthz inside the pod to test
☑ Use kubectl describe pod to see probe failures
☑ Set failureThreshold, timeoutSeconds, periodSeconds smartly
☑ Avoid overlapping readiness and liveness logic

⚙ Self-Healing in Kubernetes

Kubernetes uses a mix of:

- 🛑 **Probes**: to detect failures
- 🔁 **RestartPolicy: Always**: to auto-restart pods
- ⚖ **ReplicaSets**: to ensure desired number of pods
- 📈 **HPA (Horizontal Pod Autoscaler)**: to scale pods based on CPU/memory

🔄 Horizontal Pod Autoscaler (HPA)

"Scale pods automatically based on usage"

```
   apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler metadata:
    name: noteapp-hpa
   spec:
 scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: noteapp
    minReplicas: 2
 maxReplicas: 10
    metrics:
    - type: Resource
     resource:
   name: cpu
      target:
    type: Utilization
       averageUtilization: 50
```

📉 Common Issues

Issue	**Cause**
App receives traffic too early	Readiness probe missing or misconfigured
Pod restarts frequently	Liveness probe failing due to short timeout/delay
HPA not scaling	Missing metrics-server or low CPU usage

Issue	Cause
Probes fail randomly	Application doesn't respond consistently or at all

✅ Real-World Best Practices

- Always **start with Readiness Probe**, then add Liveness
- Use **Startup Probe** for apps that take >20s to initialize
- Test probe endpoints **locally** before production
- Use **metrics-server** or Prometheus Adapter for HPA
- Set **resource requests and limits** — HPA won't work without them!

🔁 What "Auto-Healing" Really Looks Like:

1. App crashes
2. Liveness probe fails
3. Kubernetes restarts container
4. Readiness probe holds traffic
5. App starts up
6. Traffic resumes
7. HPA adds more pods as load increases

And all this… happens without you doing anything manually.
This is how Kubernetes saves engineers from 2 AM incidents.

4. Kubernetes Networking Essentials

🌐 *If your services can't talk to each other, nothing else matters.*

🎯 Why You Must Master This

As a Kubernetes Admin, it's not enough to run pods.
You must ensure **connectivity, discoverability, and secure communication** between:

Pod ↔ Pod

- Pod ↔ Service
- External ↔ Internal
- Ingress ↔ Backend

A broken network = a broken app.

🧱 Kubernetes Networking Building Blocks

1. Pod-to-Pod Communication

- Every pod gets its **own IP address**
- All pods in the same cluster can talk to each other **without NAT**
- Powered by the **CNI plugin** (like Calico, Cilium, Flannel)

☑ Use:

```
kubectl exec -it <pod-name> -- curl <another-pod-ip>:<port>
```

🧠 In production: You'll use **Service names** instead of IPs for stability.

2. ClusterIP Services (Default)

"Stable internal DNS name for a set of pods"

- Accessed only from inside the cluster
- Load balances across healthy

endpoints spec:

```
 type: ClusterIP
 selector:
   app: noteapp
 ports:
 - port: 80
   targetPort: 8080
```

☑ Access from another pod:

3. NodePort Services

"Expose service on a static port on each worker node's IP"

```
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30007
```

☑ Access via:
http://<node-ip>:30007

⚠ Not great for production — hard to scale/load balance.

4. LoadBalancer Services

"Expose your app to the internet (cloud only)"

- Provisioned by your cloud provider (AWS ELB, GCP LB, etc.)
- Automatically routes external traffic to your pods

☑ Best for external APIs or web apps.

5. Ingress

"Single entry point for all HTTP traffic"

- Exposes multiple services via host/path rules
- Requires an Ingress Controller (e.g., NGINX,

Traefik) rules:

```
- host:
  noteapp.example.com http:
    paths:
```

```
  - path: /
    backend:
      service:
        name: noteapp
        port:
          number: 80
```

✅ Features:

- TLS (HTTPS) termination
- URL-based routing
- Rate limiting, header manipulation (via annotations)

🔧 Real-World Use Case

🎯 Scenario: You deploy 3 microservices.
Here's how traffic flows:

```
User (Browser)
   ↓
Ingress (NGINX) → /api → Backend Service
                → /cart → Cart Service
                → /auth → Auth Service
   ↓
Service → Pod(s) → App Container
```

🔍 Common Networking Debugging Scenarios

Problem	What to Check First
Pod can't reach another pod	CNI plugin status, DNS resolution
Curl to service name fails	Service selector labels match? Endpoints

Problem	What to Check First
	exist?
Ingress returns 404	Host/path config in Ingress correct?
NodePort works, but not LoadBalancer	Is cloud provider LoadBalancer controller working?
DNS lookup fails in pod	nslookup, check CoreDNS pods, configmap

🧠 Pro Tips for Admins

☑ Use kubectl get endpoints to verify service-to-pod link
☑ Use tcpdump inside pod for network tracing
☑ Always define readinessProbes to avoid routing to unhealthy pods
☑ Use **headless services** for StatefulSets (like databases)
☑ Secure traffic using **NetworkPolicies**

Test DNS and Service Routing Inside a Pod

```
kubectl exec -it <pod> -- sh

apk add curl bind-tools  # for Alpine

nslookup myservice.default.svc.cluster.local

curl http://myservice:80
```

🧠 Production Admin Checklist:

- ☐ You can explain the difference between ClusterIP, NodePort, LoadBalancer
- ☐ You can debug DNS resolution issues
- ☐ You know when to use Ingress vs LoadBalancer
- ☐ You can verify if a service has healthy endpoints
- ☐ You've secured pod-to-pod traffic with NetworkPolicies

5. Storage and Stateful Workloads

💾 *Stateless apps scale, but stateful apps run your business.*

🎯 Why This Matters

Running a frontend app in Kubernetes is easy. But when you're managing:

- 🗃 Databases
- 🧠 Caches

- 🗁 File uploads
- 📈 Queues

…you need **storage that survives restarts, reschedules, and failures.**

This section teaches you how to manage **StatefulSets, PVCs, Volumes**, and solve real issues like **PVC binding failures**, **data loss**, and **pod affinity**.

🧱 Key Concepts to Understand

1. Volumes

"Temporary or permanent storage attached to a pod"

- Volumes in K8s are **independent of containers**
- Data survives container restarts **but not pod deletion** (unless it's backed by persistent volume)

Types:

- emptyDir: Ephemeral (cleared when pod dies)
- hostPath: Uses node's local filesystem (not portable)
- persistentVolumeClaim: Used for durable storage

2. PersistentVolume (PV) and PersistentVolumeClaim (PVC)

- **PV**: Represents actual storage (EBS, NFS, etc.)
- **PVC**: User-defined request for storage

Kubernetes matches PVC → PV automatically (via StorageClass)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: ebs-sc
```

3. StorageClass

"Defines how storage is provisioned dynamically"

- You don't need to create PVs manually
- Use cloud-native provisioners (e.g., AWS EBS, GCE

PD)
```
provisioner: kubernetes.io/aws-ebs
volumeBindingMode: WaitForFirstConsumer
```

🔁 StatefulSets: What Makes Them Different

Deployment	**StatefulSet**
Pods are interchangeable	Pods are sticky (0,1,2...)
No stable identity	Each pod has persistent hostname/IP
Ephemeral by default	Attached to unique PVCs

🎯 Use StatefulSets when:

- Running **MongoDB, PostgreSQL, MySQL, Kafka**
- You need stable network identities
- Each pod needs a unique volume

Example: MongoDB with StatefulSet

```
volumeClaimTemplates:
```

```
- metadata:
    name: mongo-data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
    storageClassName: ebs-sc
```

K8s creates:

- mongo-0 → PVC: mongo-data-mongo-0
- mongo-1 → PVC: mongo-data-mongo-1

☑ Each pod gets its **own volume**
☑ Even after pod restart, data is preserved

⚠ Common Issues & Fixes

Problem	**Cause**		
PVC stuck in Pending	StorageClass missing or invalid		
Pods don't get rescheduled on same node	ReadWriteOnce restricts volume to 1 node — use anti-affinity rules		
Volume lost on pod delete	Used emptyDir instead of PVC		
Node stuck in Terminating state	PVC cleanup not complete / pod not evicted properly		

🧠 Pro Admin Practices

☑ Use **volumeBindingMode: WaitForFirstConsumer** to ensure correct zone binding
☑ For multi-zone clusters (like AWS), use **zonal EBS volumes** and set **pod affinity**

☑ Define resources.requests.storage accurately — don't over-provision
☑ Clean up **orphaned PVCs** to avoid storage leaks
☑ For shared storage (NFS), use **ReadWriteMany** access mode (rare in cloud)

🔧 Sample Troubleshooting Flow

❌ Pod pending due to unbound PVC

1. kubectl get pvc → check status
2. kubectl describe pvc → check for StorageClass issue
3. Validate storageClassName and volumeBindingMode
4. Check cloud console: is dynamic provisioner working?
5. Fix and re-apply — use helm upgrade or kubectl patch

📦 Production Tip:

- For PostgreSQL, MongoDB, or MySQL:
  - Use **StatefulSet**
  - Use **headless service** for stable DNS
  - Use **init containers** to wait for readiness

📚 You are now ready to:

- Build persistent databases in Kubernetes
- Debug PVC binding issues
- Handle stateful app deployment and recovery
- Protect against data loss during updates

6. RBAC & Access Control

🔐 *If everyone can do everything, you've already failed as an admin.*

🎯 Why RBAC Matters

Kubernetes is powerful. But with power comes risk.

If you skip proper access control:

- A dev might delete the production namespace by accident
- A CI/CD pipeline might overwrite resources it shouldn't
- Attackers can escalate privileges from compromised pods

RBAC (Role-Based Access Control) ensures that every user, pod, and tool only gets access to what they *need*, and nothing more.

🧱 Key Building Blocks of RBAC

- **1. ServiceAccounts**

"Identities for pods or processes inside the cluster"

Every pod runs as a ServiceAccount.
Default is default, but in production you should **always create dedicated accounts.**

```
serviceAccountName: deploy-bot
```

2. Roles & ClusterRoles

- **Role** → Namespace-scoped
- **ClusterRole** → Cluster-wide (can also be used in

namespace)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
namespace: dev
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

3. RoleBinding & ClusterRoleBinding

- Bind **users/serviceaccounts/groups** to roles
- RoleBinding = namespace-only
- ClusterRoleBinding = applies cluster-

wide

```
kind: RoleBinding
metadata:
  name: read-pods
    namespace: dev
```

```
subjects:
- kind: ServiceAccount
name: deploy-bot
namespace: dev
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Practical Scenario

🎯 You have a CI/CD tool (like Jenkins) deploying to dev namespace.
You want it to:

- Create deployments
- List pods
- Nothing else

☑ Create a ServiceAccount → Create Role → Bind it → Use SA in pod

🔍 Must-Know CLI for RBAC Debugging

Command	What It Does
kubectl auth can-i	Checks if a user/SA can perform an action
kubectl describe rolebinding	Shows binding details
kubectl get roles --all-namespaces	Lists all roles
kubectl get clusterrolebindings	Lists cluster-wide bindings

☑ Example:

```
kubectl auth can-i create pods --as=system:serviceaccount:dev:deploy-bot
```

Common RBAC Mistakes

Mistake	Fix
Pods use default ServiceAccount	Always define custom SA for apps/tools
ClusterRole used when Role was enough	Use least privilege — start with Role
No namespace set in RoleBinding	RBAC fails silently — always check metadata
Forgetting to set serviceAccountName	Pod uses default — won't have desired permissions

RBAC Best Practices

- ☑ **Always isolate environments** via namespaces
- ☑ **Group permissions by purpose** (read-only, deployer, admin)
- ☑ **Use automation to create roles and bindings (Helm, Terraform)**
- ☑ **Audit regularly** using tools like rakkess, who-can, and kubectl auth can-i
- ☑ **Do not bind ClusterRoles to users unless absolutely necessary**

Tool Spotlight

Tool	What It Does
rakkess	Shows who has access to what in a namespace
who-can	Reverse lookup: who can perform an action?
OPA/Gatekeeper	Policy enforcement for RBAC, resources, labels

As a Kubernetes Admin, your goal is **not just to make things work**, but to **make sure only the right people and services can do the right things** — **nothing more, nothing less**.

☑️ 7. Security Hardening Must-Knows

🛡️ *If it's not locked down — assume it's vulnerable.*

🎯 Why This Section Matters

You can't protect what you don't understand.
Most beginners stop at RBAC — but a real production admin must secure:

- Containers 🐳
- Network 🌐
- Secrets 🔐
- Access Paths 🚪
- Supply Chain 🔗

In Kubernetes, **one misconfigured YAML** can expose your entire cluster.

🧱 Top Areas to Harden in a Kubernetes Cluster

1. Pod Security Context

"Don't let your container act like root on the host."

Use these **in every pod spec**:

```
securityContext:

runAsNonRoot: true
```

```
 runAsUser: 1000
 allowPrivilegeEscalation: false
 readOnlyRootFilesystem: true
```

☑ Enforce:

- No root user
- No privilege escalation
- Immutable containers
- Sandboxed processes

o **2. Drop Linux Capabilities**

Containers don't need full OS-level powers.

```
capabilities:
  drop:
    - ALL
```

☑ If your app doesn't need it, drop it.
☑ Prevents use of tools like tcpdump, mount, chown, etc.

3. Use PodSecurityAdmission or OPA/Gatekeeper

Since Kubernetes v1.25, PSP is deprecated. Use:

- PodSecurityAdmission → applies "restricted", "baseline", "privileged" profiles
- **Gatekeeper** → policy-as-code framework (powered by Open Policy Agent)

☑ Example: Block all pods using hostNetwork: true

4. Network Policies

Control which pods can talk to whom

By default:
➡️ **Everything can talk to everything** inside the cluster 😱

✅ Create deny-all default, then allow specific communication:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress: []
```

✅ Then explicitly allow traffic by labels, ports, and namespace.

5. Secrets Management

✅ Avoid putting secrets in plain YAML or environment variables
✅ Use tools like:

Tool	Highlights
Kubernetes Secrets	Base64 encoded, not encrypted by default ⚠️
Sealed Secrets	Encrypted secrets in GitOps workflows
HashiCorp Vault	Dynamic secrets, policies, auditing ✅
External Secrets Operator	Sync secrets from cloud managers

✅ Always enable encryption at rest for Secrets in etcd (EncryptionConfiguration)

6. TLS Everywhere

Encrypt traffic between components and services

☑ Enable https on all ingress controllers
☑ Use cert-manager to manage TLS certs
☑ Use mutual TLS (mTLS) with Istio or Linkerd for service-to-service security

7. Restrict Host Access

Never allow pods to access host namespaces unless required

Block these:

hostPID: false

hostIPC: false

hostNetwork: false

Also avoid:

- hostPath volumes
- privileged: true containers

8. Supply Chain & Image Security

☑ Scan all images for vulnerabilities:

- **Trivy**, **Grype**, **Anchore**, **Snyk**

☑ Best practices:

- Use **slim base images** (python:3.11-slim)
- Pin image versions (myapp:1.2.3)
- Never use :latest in production
- Set resource limits to prevent DoS

9. Audit Logs + Logging Access

☑ Enable Kubernetes audit logging
☑ Log access to:

- API server

- Sensitive resources (secrets, pods/exec, pods/portforward)

  ✅ Use Falco or Audit2RBAC for real-time threat detection

10. Security Tools & Checks

Tool	**Purpose**
kube-bench	Run CIS benchmark checks
kube-hunter	Simulate attacks on your cluster
Trivy	Scan images & IaC for CVEs
Kyverno	Policy engine for YAML hygiene
OPA	Policy-as-code framework

🧠 Admin Best Practices

✅ Bake security into **Helm charts**, not just deployments
✅ Audit all ClusterRoleBindings periodically
✅ Never allow external access to etcd
✅ Rotate ServiceAccount tokens and secrets
✅ Train devs to understand basic security contexts

💡 **Production-grade Kubernetes isn't about how fast you deploy — it's about how little damage an attacker can do if they get in.**

✅ 8. Monitoring, Logging & Observability

Production doesn’t break silently — it whispers. Learn to hear it.

🎯 Why This Matters

Kubernetes is distributed.
Failures are **subtle**, **intermittent**, and often **invisible** — unless you’re monitoring proactively.

A production-grade K8s admin must ensure:

- Metrics are scraped
- Logs are collected
- Dashboards are useful
- Alerts are

actionable Let’s break that

down.

🧱 The Observability Pillars in Kubernetes

1. Monitoring (Metrics)

Understand what the system is doing, numerically.

✅ Tools:

- **Prometheus**: metric collector and time-series database
- **kube-state-metrics**: exposes cluster object metrics
- **node-exporter**: node-level metrics (CPU, memory, disk)

- **Grafana**: beautiful dashboards + alerts

Key Metrics to Track:

Layer	Metrics
Node	CPU/mem usage, disk I/O, pressure
Pod	Restarts, CPU/mem usage, probe failures
Deployment	Replica count, availability, rollout status
Network	Latency, drops, DNS errors

2. Logging (Events & Traces)

Understand what the system did, and why.

☑ Tools:

- **Loki** + **Promtail** → log aggregation
- **Fluent Bit** → lightweight log forwarder
- **Elasticsearch + Kibana** (EFK stack)

☑ Patterns:

- Collect logs from stdout/stderr of containers
- Enrich logs with pod labels, timestamps, severity
- Enable per-service log filters (via Loki's LogQL)

☑ Queries:

```
{app="noteapp"} |= "ERROR"

{container="nginx"} |~ "5[0-9]{2}"
```

3. Alerting

Know what broke — before your users do.

☑ Tool:

- **Alertmanager** (integrates with Prometheus)

☑ Common Alerts:

- Pod restart count > X

CPU usage > 80%

- Disk space < 10%
- Ingress latency spike
- Deployment not ready in 5 min

☑ Destinations:

- Slack
- PagerDuty
- Email

4. Dashboards That Matter

Use **Grafana** dashboards to visualize:

- Node resource usage
- Pod status
- API server request rate
- etcd health
- Network traffic patterns

☑ Grafana supports **variables**, **templating**, and **multi-datasource** panels.

☑ Start with:

- kubernetes-cluster-monitoring.json
- node-exporter-full.json

Find them on: grafana.com/grafana/dashboards

🧠 Production-Level Setup (Quick Overview)

1. **Install kube-prometheus-stack via Helm**:

```
helm install monitoring prometheus-community/kube-prometheus-stack
```

2. **Install Loki for logs**:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm install loki grafana/loki-stack
```

3. **Access Grafana**, import dashboards, configure alerts

⚠ Common Mistakes

Mistake	**Fix**
Too many alerts (alert fatigue)	Prioritize severity, group alerts
Logs not collected from all pods	Check Fluent Bit/Promtail configs, label selectors
Metrics missing	Check scrape configs and exporters
Using multiple dashboards per microservice	Consolidate into app-focused views
Alert triggers on false positives	Use average over time, not raw spike metrics

Troubleshooting Cheatsheet

Symptom	**What to Check First**
Grafana shows "No Data"	Prometheus scrape config, data source link
Logs missing from new pods	Promtail/FluentBit not matching labels or namespaces
Pod metrics blank	Missing kubelet permissions or kube-state-metrics
Alerts not firing	Alertmanager config, receiver rules, silences

🚀 Pro Tips

☑ Use **LogQL filters** in Grafana dashboards to show errors over time
☑ Alert on **rates and percentages**, not absolute numbers
☑ Export dashboards via JSON and store in Git for version control
☑ Enable **dashboards per team/service**, not per tool
☑ Include **release/version labels** in logs to correlate issues with deployments

When you master observability, you move from **reactive debugging** to **proactive reliability**.

☑ 9. Deployments, Upgrades & Rollbacks

🚀 *Move fast, don't break production.*

🎯 Why This Section Matters

Your app will change.
Your configs will evolve. Your infra will grow.

A Kubernetes Admin must ensure that:

- Deployments don't disrupt users
- Rollouts are traceable and observable
- Rollbacks are clean and fast
- Changes are reversible and repeatable

This is the art of **progressive delivery** — made reliable.

🧱 1. Deployments: The Foundation of Rollouts

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
    maxSurge: 1
```

```
maxUnavailable: 1
  selector:
    matchLabels:
      app:
  frontend
  template:
    metadata:
      labels:
        app:
    frontend spec:
      containers:
      - name: web
        image: myapp:v1.2.3
```

✅ maxSurge = how many extra pods you can spin up
✅ maxUnavailable = how many old pods can go down during update
✅ Default strategy: **RollingUpdate**
✅ For blue/green or canary — use **Argo Rollouts**, Flagger, or custom Ingress split

🔁 2. Upgrades: When You Want to Change Something

Upgrading can be:

- App version update
- Config change
- Resource adjustment
- Environment variable injection

Use GitOps tools (ArgoCD, Flux) or Helm for:

- Version tracking
- Auditable diffs

Controlled rollouts

🔧 **Real-World Tip:**

✅ Use kubectl rollout status deployment/<name> to watch progress
✅ Annotate deployments with Git commit hash:

```
metadata:
  annotations:
    app.kubernetes.io/version: "v1.2.3"
```

🧪 3. Rollbacks: When You Break Something (And You Will)

🔁 **Native rollback:**

```
kubectl rollout undo deployment/frontend
```

You can also:

```
kubectl rollout undo deployment/frontend --to-revision=2
```

✅ Kubernetes **stores revisions** of your deployments
✅ Doesn't store entire history of ConfigMaps or Secrets → use versioning manually or via Helm

4. Observability During Rollouts

- Use **readinessProbes** to hold traffic to new pods
- Monitor metrics like:
  - kube_deployment_status_replicas_updated
  - http_error_rate, latency, etc.

🔥 Never rely on "kubectl apply" alone — it tells you nothing about rollout health

🧠 5. Helm Upgrades & Rollbacks

```
helm upgrade my-release ./chart
helm rollback my-release 2
```

☑ Store your Helm values in Git
☑ Always run helm diff before applying
☑ Use --atomic to auto-rollback on failure:

```
helm upgrade --install my-release ./chart --atomic
```

⚠ Common Deployment Pitfalls

Mistake	**Consequence**	**Fix**
No readiness probe	Users hit broken pod during rollout	Add readinessProbe
Used :latest image	Untraceable, inconsistent rollout	Always pin versions (v1.2.3)
No rollback config	Manual recovery needed	Use rollout undo, Helm rollback
Applied configmap/secret without versioning	App restarts with stale config	Version and reapply with force

☑ Admin Best Practices

☑ Treat every deployment as **a risk** and **design for rollback**
☑ Use labels/annotations for Git tracking
☑ Validate config changes with kubectl diff or helm diff
☑ Monitor rollout events live using:

```
kubectl rollout status deployment/myapp
kubectl get events --sort-by='.metadata.creationTimestamp'
```

☑ For zero-downtime + progressive rollout:

- Use Argo Rollouts for canary and blue/green
- Use Ingress + header-based routing for A/B testing

When you control rollouts, you **control risk**.

10. Think Like an SRE — Resilience, Cost, and Chaos

🧠 *Your job isn't to make sure it runs. Your job is to make sure it survives anything.*

🎯 Why This Is the Final Step

You've learned how to:

- Deploy ✅
- Secure ✅
- Observe ✅
- Rollback ✅

But production-grade admins do more.
They **build systems that recover**, **scale smartly**, and **fail gracefully**.

This is the SRE (Site Reliability Engineering) mindset.

🛡 1. Design for Failure — Not Just Success

❌ Don't assume: "This will work."
✅ Assume: "This will eventually break. When it does, what happens?"

🧠 Real-World Practices:

- Use **readiness gates** to delay promotions of broken builds
- Design **multi-zone workloads** (anti-affinity)
- Setup **priorityClasses** for critical workloads
- Use **PodDisruptionBudgets (PDBs)** to protect availability during node maintenance

💸 2. Cost Optimization Without Sacrificing Reliability

Kubernetes can **waste a ton of money** if you're not careful.

✅ **Techniques:**

Right-size pods with requests and limits

- Use **HPA** to scale based on real load
- Use **Cluster Autoscaler** to optimize nodes
- Use **Spot Instances** (with taints) for non-critical workloads
- Monitor **resource usage vs. requests** over time

Tool: OpenCost
→ Get real-time cost visibility per namespace, deployment, label

🔥 3. Chaos Engineering: Test Before It Breaks

"Hope is not a strategy."

Practice **controlled failure** to:

- Prove resilience
- Validate alerts
- Test fallback logic

✅ Tools:

- **Chaos Mesh**
- **Litmus Chaos**
- **Gremlin** (SaaS)

Examples:

- Kill a pod mid-traffic
- Simulate node failure
- Introduce 100ms network latency
- Block access to a database temporarily

💡 4. Runbooks, SOPs, and Incident Readiness

You're not just building infra — you're preparing for incidents.

✅ Every production cluster should have:

Runbooks: How to fix common failures

- 🗒 **SOPs**: How to scale, backup, recover
- 🛠 **Manual failover docs**
- 📞 **Escalation process** (Slack + PagerDuty + Email)

🧠 5. Metrics That Really Matter

Anyone can graph CPU — but SREs track **SLIs/SLOs**:

Metric	Why It Matters
Latency	Are we fast enough for users?
Error rate	Are users hitting failures?
Availability	Are we up when they need us?
Saturation	Are we near resource limits?

☑ Use Prometheus or Datadog + SLIs
☑ Publish weekly SLO reports

💬 6. Communication Skills = Critical

The best admins don't just solve problems.
They **communicate calmly**, **log clearly**, and **share transparently**.

☑ During incidents:

- Use Slack threads for incident tracking
- Timestamp events
- Assign roles (commander, scribe, doer)

☑ After incidents:

- Write blameless postmortems
- Document timelines
- Extract lessons and fix the root cause

☑ **You've Reached Production-Grade Status When:**

- You **test before deploying**, not after breaking
- You **observe everything**, and ignore nothing
- You design for **failure**, not just happy paths
- You **secure by default**, not by exception
- You **explain your system** as well as you build it
- You keep the cluster **running smoothly**, while staying **calm under chaos**