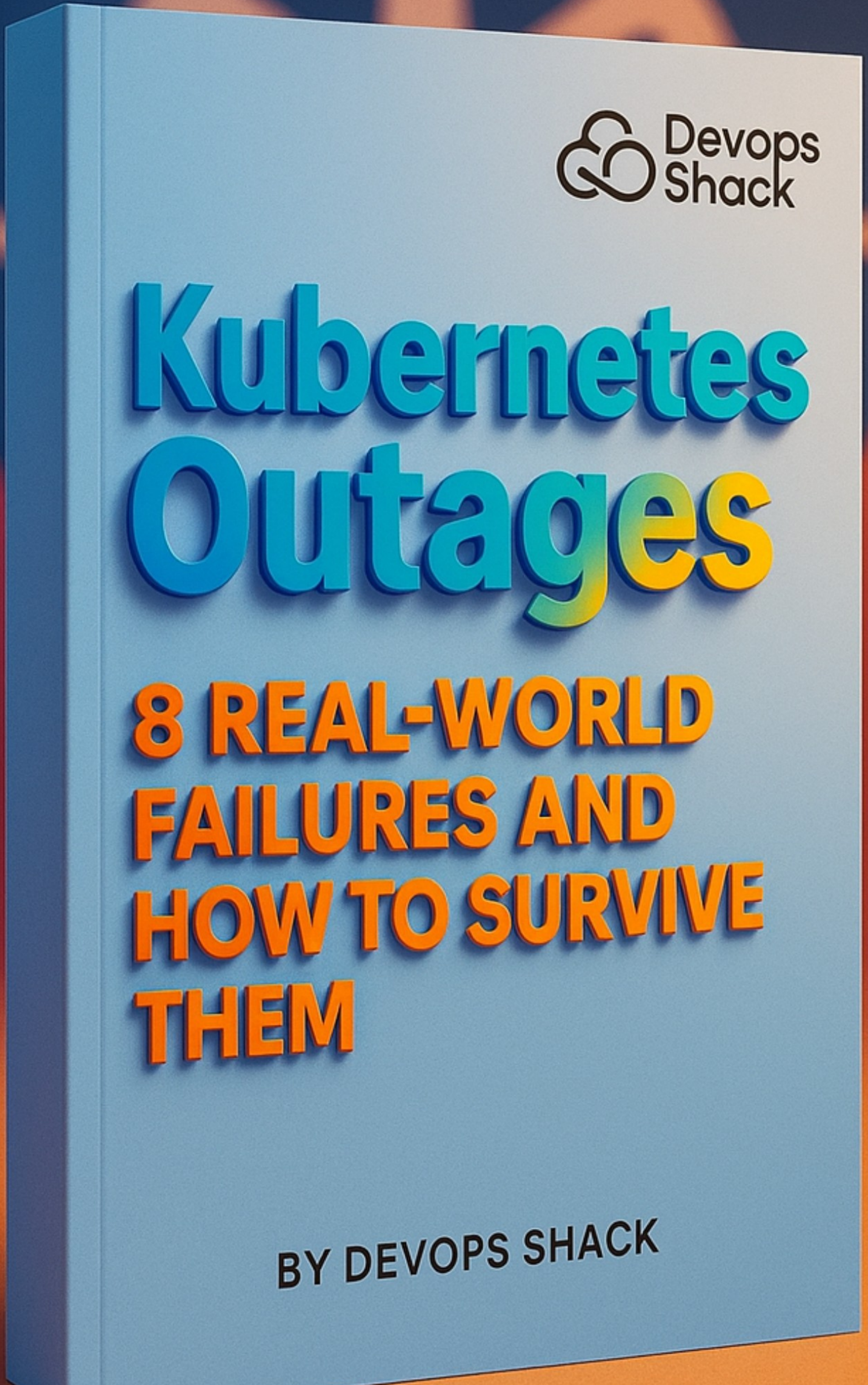

Devops Shack

Kubernetes Outages

8 REAL-WORLD FAILURES AND HOW TO SURVIVE THEM

BY DEVOPS SHACK

DevOps Shack

Kubernetes Outages: 8 Real-World Failures and How to Survive Them

🛑 *From minor misconfigurations to catastrophic cluster-wide failures — learn how to detect, diagnose, and recover like a production-grade Kubernetes Admin.*

📝 Table of Contents

Each section includes:

- 🔍 **Root Cause**
- 🛠️ **How to Diagnose**
- ✅ **Resolution Steps**

✅ 1. CrashLoopBackOff Storms

When one pod goes down, then another, then another...

- 🔍 Root Cause: Misconfigured probes, fatal exceptions, missing configs
- 🛠️ How to Diagnose: kubectl describe, kubectl logs, check events
- ✅ Resolution Steps: Fix env/configs, adjust probe timings, restart with rollback

✅ 2. Node-Level Failures or Evictions

Pods suddenly vanish or restart — your app feels flaky

- 🔍 Root Cause: Node disk full, memory pressure, CPU starvation, kubelet down
- 🛠️ How to Diagnose: kubectl get nodes, kubectl describe node, dmesg, kubelet logs

- ☑ Resolution Steps: Cordon/drain, free space, scale up, tune requests/limits

☑ 3. PersistentVolume Claim (PVC) Issues

Pods get stuck in "Pending" or apps lose data on restart

- 🔍 Root Cause: StorageClass misconfig, volume zone mismatch, PV unavailable
- 🛠 How to Diagnose: kubectl get pvc, describe pvc, look for unbound claims
- ☑ Resolution Steps: Fix SC, verify EBS/GCE volumes, use volumeBindingMode: WaitForFirstConsumer

☑ 4. Ingress or LoadBalancer Downtime

The app works, but nobody can reach it

- 🔍 Root Cause: Broken Ingress rules, SSL cert expired, DNS misrouting, LoadBalancer misprovisioned
- 🛠 How to Diagnose: kubectl describe ingress, cert-manager logs, dig/DNS tools
- ☑ Resolution Steps: Fix ingress path/host, re-issue certs, re-provision LoadBalancer, check annotations

☑ 5. DNS Resolution Failures Inside Cluster

Pods can't reach services via name — only by IP

- 🔍 Root Cause: CoreDNS crash, ConfigMap issues, network plugin problem
- 🛠 How to Diagnose: Check coredns pod status/logs, run nslookup, dig in affected pods
- ☑ Resolution Steps: Restart CoreDNS, validate ConfigMap, upgrade CNI plugin, clear stuck pods

✅ 6. Etcd Data Corruption or Unavailability

API server becomes unstable or unresponsive

- 🔍 Root Cause: Disk pressure, snapshot failure, network split, cluster scaling bug
- 🛠 How to Diagnose: Check etcd logs, API latency spikes, use etcdctl for cluster health
- ✅ Resolution Steps: Restore from snapshot, clean etcd data dir, scale etcd properly, isolate disk issues

✅ 7. API Server Saturation or Hang

Cluster seems frozen — no pods list, no deployments work

- 🔍 Root Cause: API server overwhelmed, controller-manager misbehaving, cert expiry
- 🛠 How to Diagnose: Check API server metrics, latency, controller logs, Prometheus alerts
- ✅ Resolution Steps: Scale control plane, add resource limits, clean long-running watches

✅ 8. Broken Rollouts / Bad Deployments to Prod

Everything was working… until you deployed

- 🔍 Root Cause: Misconfigured image, failing containers, missing env variables, bad YAML
- 🛠 How to Diagnose: kubectl rollout status, pod logs, probe failures, image digests
- ✅ Resolution Steps: Use rollout undo, implement canary strategy, test configs in staging

✅ 1. CrashLoopBackOff Storms

🛑 *"Everything was fine… and then pods just started crashing — over and over."*

🔍 Root Cause

The CrashLoopBackOff status means that a container **starts → crashes → Kubernetes restarts it → crashes again** — in a loop.

In large clusters, this can quickly become a **storm** if multiple pods, deployments, or namespaces are affected simultaneously.

⚠ Common Causes:

1. **Invalid or missing environment variables**
   - App depends on secrets/configs that aren't loaded
   - Example: DB connection string is missing or wrong
2. **Faulty or misconfigured probes**
   - Liveness or readiness probe kills the container before it's ready
   - Too short initialDelaySeconds, overly strict timeouts
3. **Fatal application bugs**
   - Application throws unhandled exception on boot
   - Segfault, null pointer, syntax error in Python, etc.
4. **Bad image or command override**
   - Wrong startup script in Dockerfile or pod spec
   - App exits immediately after starting
5. **Permission errors**
   - App tries to write to read-only filesystem
   - Non-root user lacks access to required directories
6. **OOMKilled (Out Of Memory)**
   - Pod exceeds memory limits and is killed by kernel
   - Looks like CrashLoopBackOff but root cause is resource starvation

🛠 How to Diagnose

◇ Step 1: kubectl get pods -n <namespace>

```
NAME                        READY   STATUS             RESTARTS   AGE
frontend-67c7857bbd-jx97q   0/1     CrashLoopBackOff   5          2m50s
```

Look at the number of **restarts** — if it's going up rapidly, you're in a loop.

◇ Step 2: Describe the Pod

```
kubectl describe pod frontend-67c7857bbd-jx97q -n <namespace>
```

Look for:

- **Last State → Terminated**
- **Reason: OOMKilled / Error**
- **Events** at the bottom, especially related to:
  - Probes failing
  - Image pull success/failure
  - Container exit codes

📍 Example output:

```
Last State: Terminated
Reason: Error
Exit Code: 1
Message: fatal: unhandled exception...
```

◇ Step 3: Get Logs

```
kubectl logs <pod-name> -n <namespace>
```

If the pod restarts quickly, use:

```
kubectl logs <pod-name> -n <namespace> --previous
```

Look for:

- Stack traces

- Config file not found
- Permission denied
- Connection refused
- Syntax errors

◇ Step 4: Exec into the Pod (if it stays alive)

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

Check:

- File paths
- Permissions
- Missing files or misconfigured environment variables
- Network access to DBs or services

☑ Resolution Steps

Here’s how you recover depending on the root cause:

🔧 Scenario 1: Missing Config or Secret

- Use kubectl describe deployment <name> and check envFrom, valueFrom, or secretKeyRef fields
- Ensure the Secret or ConfigMap exists

```
kubectl get secret | grep my-app-secret
kubectl get configmap | grep config
```

☑ Fix:

- Recreate the missing config
- Re-apply the deployment
- Use default values where possible to avoid hard crashes

🔧 Scenario 2: Bad Probes

- If probes are too aggressive, the pod may get killed before it's fully ready

☑ Fix:

- Increase initialDelaySeconds, timeoutSeconds, and failureThreshold

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 20
  periodSeconds: 5
  timeoutSeconds: 2
  failureThreshold: 5
```

- Temporarily remove liveness/readiness probes to stabilize, then reintroduce slowly

🔧 Scenario 3: App Bug or Crash

- Fix the bug in code or startup script
- Rebuild and push the Docker image
- Redeploy with updated tag (never use :latest)

```
docker build -t myapp:v1.2.3 .
docker push myapp:v1.2.3
kubectl set image deployment/myapp myapp=myapp:v1.2.3
```

🔧 Scenario 4: OOMKilled

- Describe pod and look for:

```
Last State: Terminated
Reason: OOMKilled
```

☑ Fix:

- Increase memory limit in spec:

```
resources:
 requests:
  memory: "256Mi"
 limits:
  memory: "512Mi"
```

- Optimize app memory usage
- Monitor with Prometheus/metrics-server

🔧 Scenario 5: Command Error

- Check for command, args, or entrypoint issues

☑ Fix:

- Remove unnecessary overrides in the Deployment YAML
- Test locally using:

```
docker run -it myapp:v1.2.3 /bin/sh
```

🔄 Stabilization Tactics

☑ Scale Down the Broken Deployment Temporarily

```
kubectl scale deployment <name> --replicas=0
```

☑ Roll Back to Last Known Good State

```
kubectl rollout undo deployment <name>
```

☑ Isolate the Issue in a Staging Namespace

- Use kubectl get deployment -o yaml
- Modify and apply in a test namespace

🧠 Best Practices to Prevent CrashLoopBackOff Storms

Practice	Description
Validate configs before deployment	Use kubectl diff, kubeval, Helm dry-runs

Practice	Description
Use health checks wisely	Tune probes per container startup time
Don’t use :latest tags	Use pinned image versions
Automate smoke tests post-deployment	Use jobs, init containers, or canary checks
Always set resource requests & limits	Avoid random crashes and OOMs

🧪 Example Incident Timeline

Time	Event
10:00 AM	New image v2.0.1 deployed
10:01 AM	Probes start failing, pods restart
10:02 AM	Team sees CrashLoopBackOff on UI
10:04 AM	kubectl logs shows config missing
10:06 AM	Secret not mounted due to typo
10:08 AM	Secret fixed → redeploy → recovery

📚 Wrap-Up

CrashLoopBackOff is **the most common K8s outage** — but also **the most fixable**, if you:

- Read logs carefully
- Use describe/events effectively
- Follow rollback-first, debug-later mindset

☑ 2. Node-Level Failures or Evictions

💥 *Pods are disappearing. Apps are unstable. And nobody knows why...*

🔍 Root Cause

When nodes (i.e., Kubernetes worker machines) face issues, Kubernetes begins **evicting pods** to keep the system stable.

This leads to:

- Pods restarting or disappearing unexpectedly
- Services intermittently unavailable
- PVCs failing to mount (due to node change)
- Deployment rollouts getting stuck

These failures often seem "random" — until you realize it's **node pressure**, **kubelet crash**, or **resource exhaustion**.

⚠ Common Node-Level Outage Causes:

1. **Memory/CPU pressure**
   - Node reaches resource limits
   - Kubelet begins **evicting lower-priority pods**
2. **Disk pressure / full disk**
   - Node cannot schedule or run pods
   - Ephemeral storage or image cache uses up space
3. **Network partition**
   - Node becomes **NotReady** (lost heartbeat)
   - Cluster fails to reach kubelet
4. **Kubelet crash / misconfiguration**
   - Broken kubelet config, TLS issues
   - No control loop running → node becomes unresponsive
5. **Pod misplacement**
   - Pod is tied to volume/zone and can't be moved easily (e.g., StatefulSets with PVC)

🛠 How to Diagnose

◇ Step 1: Check Node Status

```
kubectl get nodes
```

Look for:

- NotReady → indicates loss of kubelet heartbeat
- SchedulingDisabled → node cordoned
- Taints or labels blocking scheduling

◇ Step 2: Describe the Node

```
kubectl describe node <node-name>
```

Focus on:

- **Conditions** (MemoryPressure, DiskPressure, PIDPressure)
- **Events** → evictions, kubelet errors
- **Allocated resources vs. available**

📍 Example:

```
Conditions:
  MemoryPressure     True
  DiskPressure       False
  Ready              False
Events:
  Type     Reason                Age   From                   Message
  ----     ------                ----  ----                   -------
  Warning  EvictionThresholdMet   3m    kubelet, ip-10-0-1-29   Attempting to
reclaim memory
```

◇ Step 3: Pod Events & Evictions

```
kubectl get events --sort-by='.lastTimestamp' | grep -i evict
```

You'll see something like:

```
Warning  Evicted  pod/frontend-xyz  The node was low on resource: memory.
```

◇ Step 4: Check System-Level Logs (Node OS)

SSH into the node (if possible):

```
journalctl -u kubelet
dmesg | tail
df -h
```

Look for:

- Disk full
- OOM events
- Docker/image GC logs
- Kubelet crash messages

☑ Resolution Steps

🔧 Scenario 1: Memory or CPU Pressure

Pods are being evicted to maintain node health.

☑ Fix:

- Increase node size or cluster auto-scaler
- Lower pod resource limits
- Prioritize critical workloads with **priorityClassName**

```
priorityClassName: high-priority
```

- Use **Vertical Pod Autoscaler (VPA)** to right-size containers over time

🔧 Scenario 2: Disk Pressure / Full Volume

Node fails due to full disk — usually in:

- /var/lib/docker (image cache)
- /var/log (excessive logs)

☑ Fix:

- SSH into node, clean up /var/log
- Run Docker image prune:

```
docker system prune -af
```

- Tune garbage collection thresholds in kubelet config:

```
evictionHard:
  memory.available: "100Mi"
  nodefs.available: "10%"
```

🔧 Scenario 3: Kubelet Crash / NotReady

Node enters NotReady state and pods get stuck.

☑ Fix:

- Restart kubelet on the node

```
systemctl restart kubelet
```

- Verify node joins the cluster again:

```
kubectl get nodes
```

- If permanent issue → **cordon and drain**

```
kubectl cordon <node>
kubectl drain <node> --ignore-daemonsets --delete-emptydir-data
```

Then reschedule pods elsewhere.

🔧 Scenario 4: Pod Gets Evicted and Doesn’t Reschedule

PVC stuck or pod with zone-affinity.

☑ Fix:

- Check volumeBindingMode: WaitForFirstConsumer
- Use **PodAntiAffinity** to distribute pods across zones
- Enable **StorageClass** with dynamic provisioning
- Use **StatefulSet** properly to avoid zone/PVC conflicts

🧠 Preventive Measures & Best Practices

Task	**Why It Helps**
Monitor node resource usage	Avoid silent pressure buildup
Set requests and limits for all pods	Prevent noisy neighbor syndrome
Enable Cluster Autoscaler	Adds/removes nodes based on workload
Clean image and log caches	Prevent disk pressure
Prioritize critical services	Evict low-priority pods first (not core APIs)
Spread workloads	Use podAntiAffinity and topologySpreadConstraints

🧪 Incident Timeline Example

Time	**Event**
2:00 PM	Ingress unavailable for some users
2:01 PM	API server shows node ip-10-0-1-29 as NotReady
2:03 PM	Events show memory pressure & pod evictions
2:04 PM	Team drains node, reroutes workload
2:07 PM	Cluster stabilizes after HPA and rescheduling

📑 Summary

Node issues can be stealthy but catastrophic.
Knowing how to detect, drain, debug, and reschedule gives you **real control** as a Kubernetes Admin.

☑ 3. PersistentVolume Claim (PVC) Issues

📦 *"My pod is stuck in Pending…" — aka, your app can't store or read any data.*

🔍 Root Cause

A PersistentVolumeClaim (PVC) outage occurs when:

- A pod **can't bind to a volume**
- A volume is **attached to another node**
- A pod is **stuck in Pending** or **ContainerCreating**

- A StatefulSet crashes after restart because data is **lost or inaccessible**

This affects **databases**, **caches**, **queues**, and any workload that stores **state**.

⚠ Common Causes of PVC-Related Outages

1. **StorageClass misconfiguration**
    - StorageClass name typo
    - Non-existent or deprecated provisioner
2. **Zone/Availability Mismatch**
    - Pod scheduled in a zone that can’t access the volume
    - Common in AWS/GCP with zonal storage
3. **Volume is stuck in use**
    - Volume still attached to old pod/node
    - Failed to detach during rescheduling
4. **Missing or deleted PV**
    - Underlying EBS/GCE disk deleted manually
    - PV deleted while PVC still exists
5. **Pod restarted without a PVC**
    - YAML missing volumeClaimTemplates or persistentVolumeClaim

🛠 How to Diagnose

◇ Step 1: Get PVC Status

```
kubectl get pvc -n <namespace>
```

Look for:

- STATUS: Pending → not bound
- STATUS: Lost → bound PV is gone

- STATUS: Terminating → stuck deletion

◇ Step 2: Describe PVC

```
kubectl describe pvc <name> -n <namespace>
```

Check:

- **Bound Volume** (PV name)
- **StorageClass**
- **Events**
  - “waiting for a volume to be created”
  - “waiting for first consumer”

📍 Example:

```
Type    Reason           Age   From                       Message
----    ------           ----  ----                       -------
Normal  Provisioning      2m   persistentvolume-controller  External provisioner is provisioning volume
Warning  ProvisioningFailed  2m   persistentvolume-controller  StorageClass not found
```

◇ Step 3: Get Pod Status

```
kubectl get pods -o wide -n <namespace>
kubectl describe pod <name>
```

Look for:

- Volume mount issues
- Pod stuck in ContainerCreating
- “Unable to attach or mount volumes”
- Events related to volumeAttachment

◇ Step 4: Inspect the Node (if needed)

SSH into node:

```
lsblk

mount

df -h
```

Look for volumes still mounted or stale device references (esp. EBS).

☑ Resolution Steps

🔧 Scenario 1: StorageClass Misconfigured or Missing

📍 Error:

```
Warning  ProvisioningFailed  StorageClass "ebs-sc" not found
```

☑ Fix:

- Create correct StorageClass:

```
apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

  name: ebs-sc

provisioner: kubernetes.io/aws-ebs

volumeBindingMode: WaitForFirstConsumer
```

- Ensure volumeBindingMode matches your cluster zone setup.

🔧 Scenario 2: Volume Bound but Pod Can't Mount

📍 Error:

```
Failed to attach volume "pvc-xxxx" to node "ip-10-0-1-28": Timeout
```

☑ Fix:

- Volume might still be attached to old node → detach manually via cloud console
- Cordon old node:

```
kubectl cordon <node>
```

- Delete stuck pod → allow rescheduling
- Use anti-affinity to prefer same zone

🔧 Scenario 3: PVC Stuck in Pending

📍 Error:

```
waiting for first consumer
```

☑ Fix:

- Add proper **scheduler hints** (e.g., nodeSelector, affinity)
- Set volumeBindingMode: WaitForFirstConsumer (delays provisioning until pod is scheduled)
- Check if there are enough nodes in the same zone as the volume

🔧 Scenario 4: StatefulSet Restart Breaks Volume Attach

☑ Fix:

- Use volumeClaimTemplates for each replica
- Never delete StatefulSet without deleting PVCs manually

```
kubectl delete pvc data-mongo-0
```

- Use retentionPolicy: Retain only if you're intentionally persisting PVC after pod delete

🔧 Scenario 5: Deleted or Lost PV

📍 Error:

```
PVC bound to PV "pv-xyz", but PV does not exist
```

☑ Fix:

- Recreate the PV with same name, UID (if known)
- Restore from backup
- Update YAML to request a new PVC

🧠 Best Practices for PVC Stability

Practice	**Why It Helps**
Use WaitForFirstConsumer binding mode	Avoid provisioning volumes in wrong zone
Always specify StorageClass	Prevent default mismatch or fallback
Monitor volume events	Detect provisioning delays or failures
Don’t hard-delete PVs manually	Avoid PVC stuck/lost state
Tag volumes with pod info (cloud)	Easier to trace and debug in AWS/GCP

🔐 Bonus: How to Avoid Data Loss in Stateful Apps

☑ Use **backup tools**:

- Velero for cluster + volume snapshots
- Database-level backups (mysqldump, mongodump)
- VolumeSnapshot API in Kubernetes 1.17+

☑ Automate **PVC cleanup only after validation**

☑ Use **Retain reclaimPolicy** on critical volumes:

```
reclaimPolicy: Retain
```

🧪 Sample Incident Timeline

Time	**Event**
3:00 PM	MongoDB pod stuck in Pending after node restart

Time	Event
3:02 PM	PVC unable to bind → storageClass missing
3:05 PM	Admin re-applies StorageClass, scales down StatefulSet
3:07 PM	PVC binds successfully → StatefulSet restored

🗂 Summary

Storage outages are sneaky — but often stem from simple misconfigurations.
The key to recovery is:

- Understand PVC/PV lifecycle
- Know your cloud storage behavior
- Respect zones, bindings, and reclaim policies

☑ 4. Ingress or LoadBalancer Downtime

🌐 *"The app is working... but nobody can access it."*

🔍 Root Cause

Ingress and LoadBalancer issues result in **external traffic** never reaching your services.
Internally, pods might be healthy, but **from the user's perspective — it's down.**

⚠ Common Causes of Ingress/LoadBalancer Downtime:

1. **Ingress misconfiguration**
   - Incorrect host/path rules
   - Missing or incorrect service backend

  - Wrong port mapping

2. **SSL/TLS certificate issues**
   - Expired certs
   - CertManager not renewing
   - Wrong DNS-01 challenge setup
3. **Cloud LoadBalancer provisioning failure**
   - Failed due to wrong annotations
   - Service type misconfigured (ClusterIP instead of LoadBalancer)
   - LoadBalancer stuck in Pending
4. **DNS misrouting**
   - External domain not pointed to correct LoadBalancer IP
   - DNS record TTL caching old IPs
5. **Network policy or security group blocking traffic**
   - NSGs/Firewalls deny inbound access on ports (e.g., 80/443)

🛠 How to Diagnose

◇ Step 1: Check Ingress Resource

```
kubectl get ingress -n <namespace>
```

Look for:

- ADDRESS column: Is IP/hostname assigned?
- HOSTS: Does it match expected DNS?
- TLS: Enabled?

◇ Step 2: Describe the Ingress

```
kubectl describe ingress <name> -n <namespace>
```

Check for:

- Annotations (e.g., nginx.ingress.kubernetes.io/*)

- Backend service mapping
- Events like:

```
Warning: error resolving service
```

◇ Step 3: Validate LoadBalancer Service

```
kubectl get svc -n <namespace>
```

If EXTERNAL-IP is pending for minutes:

```
kubectl describe svc <name>
```

Possible causes:

- Wrong annotations
- No cloud provider integration
- Unsupported service type (must be LoadBalancer)

◇ Step 4: Check Certificate Status (for HTTPS)

If you're using **CertManager**:

```
kubectl describe certificate -n <namespace>
kubectl describe challenge -n <namespace>
```

Look for:

- Ready: False
- Failed to complete DNS challenge
- Error creating ACME challenge

◇ Step 5: DNS Validation

Use dig, nslookup, or curl:

```
dig yourapp.example.com
```

Make sure the DNS A/AAAA record points to the **LoadBalancer EXTERNAL-IP**.

Test HTTP/HTTPS:

```
curl -v https://yourapp.example.com
```

☑ Resolution Steps

🔧 Scenario 1: Ingress Path or Host Mismatch

📍 Symptom:

```
404 Not Found from ingress-controller
```

☑ Fix:

- Verify path in Ingress spec:

```
rules:
- host: yourapp.example.com
 http:
  paths:
  - path: /
   pathType: Prefix
   backend:
    service:
     name: yourapp-svc
     port:
      number: 80
```

- Ensure Service yourapp-svc exists and is ClusterIP
- Ensure app listens on correct port (match service and container port)

🔧 Scenario 2: Stuck LoadBalancer / No External IP

☑ Fix:

- Ensure service type is correct:

```
kind: Service
spec:
  type: LoadBalancer
```

- Check cloud provider annotations:

Cloud	Example Annotation
AWS	service.beta.kubernetes.io/aws-load-balancer-type: nlb
GCP	cloud.google.com/load-balancer-type: "External"

- Recreate the service if needed:

```
kubectl delete svc <name>
kubectl apply -f svc.yaml
```

🔧 Scenario 3: SSL Certificate Expired or Failing

☑ Fix:

- Re-issue cert:

```
kubectl delete certificate <name>
kubectl apply -f cert.yaml
```

- Ensure ClusterIssuer exists and is valid
- Use Let’s Encrypt staging environment for testing:

```
issuerRef:
  name: letsencrypt-staging
  kind: ClusterIssuer
```

- Check DNS-01 challenge with CertManager logs

🔧 Scenario 4: DNS Not Updated

☑ Fix:

- Update domain provider to point A record to LoadBalancer IP
- Use CNAME for dynamic DNS like AWS ELB
- Flush DNS cache locally:

```
sudo systemd-resolve --flush-caches
```

🔧 Scenario 5: Firewall/Security Group Blocks

☑ Fix:

- Open inbound ports (80, 443) in:
  - AWS Security Groups
  - Azure NSGs
  - GCP Firewall Rules

Check NGINX ingress logs:

```
kubectl logs -n ingress-nginx -l app.kubernetes.io/name=ingress-nginx
```

🧠 Best Practices for Ingress/LoadBalancer Stability

Best Practice	**Why It Helps**
Use pathType: Prefix	Avoid path match errors
Always monitor cert-manager logs	Avoid silent certificate failures
Use DNS TTLs of 300s or less	Enable fast propagation during cutovers
Tag and track LoadBalancer IPs	Map DNS records accurately
Use external-dns to auto-manage	Automate route53/CloudDNS/GoDaddy records

🧪 Sample Outage Timeline

Time	Event
10:00 AM	Service deployed, but Ingress returns 404
10:02 AM	Ingress backend service name was typo (myapp-svcx)
10:03 AM	CertManager fails to renew cert due to DNS challenge
10:06 AM	Fixed service name and DNS TXT record → HTTPS restored
10:09 AM	Domain resolves to LoadBalancer IP → Site up

🗂 Summary

Ingress and LoadBalancer outages are common but often:

- Easily diagnosable with describe, logs, dig, and curl
- Caused by **simple misconfigurations**, not hard bugs
- Fixed by re-checking: ingress → service → cert → DNS → cloud provider

☑ 5. DNS Resolution Failures Inside the Cluster

🧠 *“My pods are running… but they can’t talk to each other.”*

🔍 Root Cause

In Kubernetes, internal DNS (usually powered by **CoreDNS**) enables service discovery.
Pods use service names like myapp.default.svc.cluster.local instead of IPs.

When DNS resolution fails, it breaks:

- Microservice-to-microservice communication
- Database lookups (db-service.default.svc)
- Init containers waiting on services
- Liveness probes that use hostnames

⚠ Common Causes of DNS Failures in Kubernetes:

1. **CoreDNS pods crash or become unschedulable**
2. **CoreDNS ConfigMap errors** (e.g., malformed forward or stubDomain)
3. **Network plugin (CNI) breaks DNS routing**
4. **Pods misconfigured or lacking DNSPolicy**
5. **loop plugin missing in CoreDNS config (causes recursive lookups)**
6. **Kubelet config breaking DNS resolution on nodes**

🛠 How to Diagnose

◇ Step 1: Check CoreDNS Pod Status

```
kubectl get pods -n kube-system -l k8s-app=kube-dns
```

Expected:

```
coredns-6d8c4cb4d-xxxx   1/1     Running
```

If status is CrashLoopBackOff, Pending, or Evicted — it's a red flag.

◇ Step 2: Check CoreDNS Logs

```
kubectl logs -n kube-system -l k8s-app=kube-dns
```

Look for:

- plugin/forward: no upstream
- unable to forward request
- loop detected
- cannot resolve...

◇ Step 3: Run DNS Test from Inside a Pod

Start a debugging pod:

```
kubectl run -i --tty dnsutils --image=busybox --restart=Never -- sh
```

Inside:

```
nslookup kube-dns.kube-system.svc.cluster.local
```

nslookup google.com

📍 If both fail: **cluster DNS is broken**
📍 If only cluster domain fails: **CoreDNS problem**
📍 If google.com fails too: **Node DNS config broken**

◇ Step 4: Check CoreDNS ConfigMap

```
kubectl get configmap coredns -n kube-system -o yaml
```

Look for:

- Incorrect forward plugin usage
- Recursive DNS loops
- Missing or malformed entries like:

```
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
      pods insecure
      fallthrough in-addr.arpa ip6.arpa
    }
    forward . /etc/resolv.conf
    cache 30
    loop
    reload
    loadbalance
}
```

☑ Resolution Steps

🔧 Scenario 1: CoreDNS Pod Down or Crashing

📍 Symptom:

```
CrashLoopBackOff - plugin/forward: no upstream
```

☑ Fix:

- Check if CoreDNS can't reach upstream resolver:
  - /etc/resolv.conf inside CoreDNS container might be misconfigured
- Restart CoreDNS:

```
kubectl rollout restart deployment coredns -n kube-system
```

- If using custom DNS servers, add them to the ConfigMap forward section.

🔧 Scenario 2: CoreDNS ConfigMap Errors

📍 Symptom:

```
plugin/loop: detected recursive DNS query
```

☑ Fix:

- Add loop plugin:

```
loop
```

- Always keep this **below the forward plugin** to prevent recursion
- Reapply ConfigMap and restart CoreDNS:

```
kubectl apply -f coredns.yaml
kubectl rollout restart deployment coredns -n kube-system
```

🔧 Scenario 3: CNI/Networking Plugin is Blocking DNS

☑ Fix:

- If CoreDNS is running but pods can't resolve, the issue could be **networking (CNI plugin)**:
    - Check iptables, Calico/Weave/Flannel logs
    - Restart CNI pods

```
kubectl rollout restart ds <cni-daemonset-name> -n kube-system
```

- Ensure DNS traffic is allowed between namespaces and to kube-system

🔧 Scenario 4: Pod DNSPolicy Misconfigured

☑ Fix:

- Pod should have:

```
dnsPolicy: ClusterFirst
```

This ensures DNS queries go through CoreDNS first.

- Avoid overriding dnsConfig unless you know what you're doing.

🧠 Best Practices to Prevent Cluster DNS Outages

Practice	Why It Matters
Use liveness/readiness probes for CoreDNS	Auto-recovery of DNS service
Always test nslookup and dig during deploys	Validate app-to-app communication
Backup and version your CoreDNS ConfigMap	Prevent accidental edits
Don't delete CoreDNS in dev/test by mistake	Some tools may remove it (like kind delete)
Monitor DNS resolution latency via Prometheus	Catch slow queries before failure

🧪 Sample Incident Timeline

Time	Event
9:00 AM	Apps report connection timeout errors internally
9:01 AM	DNS resolution failing for mydb.default.svc
9:02 AM	CoreDNS pods seen in CrashLoopBackOff
9:04 AM	Logs show loop plugin missing
9:05 AM	Added loop to ConfigMap, restarted CoreDNS
9:07 AM	Internal DNS restored, apps working again

🗐 Summary

Internal DNS failures are **invisible to external users** but **cripple microservices**.

The fix often comes down to:

- Ensuring CoreDNS is healthy and configured right
- Validating pod-level resolution with tools like nslookup
- Keeping your CNI and CoreDNS plugins updated and monitored

☑ 6. Etcd Data Corruption or Unavailability

🧠 *“The control plane is choking… nothing responds.”*

🔍 Root Cause

Etcd is the **key-value store** behind Kubernetes.
If it’s down or corrupted:

- API server becomes unresponsive
- Cluster state can't be read or written
- No new pods can be created
- Existing workloads may run, but can't be managed

⚠ Common Triggers

- Disk full on etcd host
- Network split between etcd peers
- Etcd crash or panic due to corruption
- Snapshot restoration gone wrong
- Wrong etcd version or flags during upgrade

🛠 Quick Diagnosis

```
kubectl get componentstatuses
```

- etcd should be Healthy. If not → investigate immediately.

SSH into master node:

```
journalctl -u etcd
```

Check for:

- panic, corrupt, or disk quota exceeded
- Connectivity issues between etcd nodes

Check etcd health manually:

```
etcdctl endpoint health
```

☑ Resolution Steps

🔧 1. Disk Full / IOPS Bottleneck

- Free up space or resize disk
- Ensure SSD-backed volume with fast IOPS

🔧 2. Corrupted DB

- Stop etcd
- Move corrupted DB:

```
mv /var/lib/etcd /var/lib/etcd.bak
```

- Restore from snapshot:

```
etcdctl snapshot restore <snapshot.db>
```

🔧 3. Quorum Lost (HA etcd)

- Needs **majority of nodes** (n/2 + 1) to be healthy
- Bring back offline peers or remove stale ones from cluster config

🔐 Prevention

Practice	**Why It Helps**
Schedule etcd snapshots daily	Enables fast recovery
Monitor etcd disk usage	Avoid quota triggers

Practice	Why It Helps
Use dedicated etcd disks	Isolate from OS/app traffic
Never kill etcd without backup	Risk of total cluster loss
Always upgrade etcd with caution	Version mismatch can corrupt

🧪 Incident Snapshot

Time	Event
2:00 PM	kubectl hangs — API server slow
2:01 PM	etcd shows disk full, panic in logs
2:05 PM	Snapshot restored, etcd restarted
2:07 PM	API back up, cluster recovered

☑ 7. API Server Saturation or Hang

“kubectl times out. Dashboards stop loading. Everything feels stuck.”

Root Cause

The **Kubernetes API server** is the brain of the cluster. If it slows or crashes:

- kubectl commands fail
- CI/CD pipelines hang
- Cluster operations break
- Monitoring tools can’t pull data

⚠ Common Causes

- **Too many requests** (e.g., thousands of kubectl/monitoring calls)
- **Excessive watch connections** (Prometheus, controllers)
- **Heavy write load** (frequent updates from custom controllers)
- **Etcd lagging or unreachable**
- **Admission webhooks timing out**

Quick Diagnosis

◇ **Step 1: Check API server status**

```
kubectl get --raw /healthz
```

Fails = unhealthy.

◇ **Step 2: View metrics (if accessible)**

```
kubectl top pods -n kube-system
```

High CPU/memory usage in kube-apiserver?

◇ Step 3: Look into logs

```
kubectl logs -n kube-system -l component=kube-apiserver
```

Look for:

- timeout, too many open files, etcd timeout, webhook errors

☑ Resolution Steps

🔧 1. Reduce Load

- Pause CI/CD pipelines
- Temporarily disable noisy monitoring agents

🔧 2. Audit & Remove Excess Watchers

- Reduce Prometheus scrape intervals
- Limit number of kubectl watches from automation

🔧 3. Scale the Control Plane

- Increase CPU/memory for API server pods (if self-managed)
- In EKS/GKE → upgrade control plane or contact support

🔧 4. Fix Etcd or Webhook Lag

- Etcd timeout = fix etcd first
- Misbehaving webhook = disable via ValidatingWebhookConfiguration

🔐 Prevention

Practice	**Why It Helps**
Set API rate limits via audit/webhook	Prevent request overload
Use HorizontalPodAutoscaler for clients	Avoid spiky loads
Keep admission webhooks lightweight	Prevent processing delays

Practice	Why It Helps
Monitor apiserver_request_duration	Catch early signs of stress

🧪 Incident Snapshot

Time	Event
6:00 PM	kubectl get pods hangs
6:02 PM	kube-apiserver logs show timeouts
6:04 PM	Paused CI jobs + throttled Prometheus
6:07 PM	API server load drops → cluster recovers

☑ 8. CrashLoopBackOff Storms

♻ *“Pods keep restarting endlessly. Nothing is stable.”*

🔍 Root Cause

This happens when pods crash repeatedly and Kubernetes **backs off** from restarting them.

Common symptoms:

- kubectl get pods shows CrashLoopBackOff
- App is never healthy
- Cluster gets overloaded if many pods are affected

⚠ Common Causes

- App code bug or misconfig
- Bad secrets/configMaps/env vars
- Missing volume mount
- Init container fails
- Liveness/readiness probes misconfigured
- Dependency service (e.g., DB) unavailable

🛠 Quick Diagnosis

```
kubectl describe pod <pod-name>
kubectl logs <pod-name> -c <container>
```

Check:

- Recent exit codes (137 = OOM, 1 = crash, 126/127 = permission issues)

- Probe failures
- Init container failures

☑ Resolution Steps

🔧 1. App Bug or Env Issue

- Check logs for stack trace
- Fix env vars, image tags, secrets
- Validate command and args

🔧 2. Probe Failures

- Comment out livenessProbe and readinessProbe temporarily
- Test basic container startup

🔧 3. Missing Volume or PVC

- Check if PVC is Pending or Lost
- Fix volume or storage class

🔧 4. Dependency Unavailable

- App waiting on DB, Redis, etc?
- Add initContainers to check connectivity before start

🔐 Prevention

Practice	**Benefit**
Add retry logic in apps	Handles boot-time flakiness
Use startupProbe for slow apps	Avoid early restarts
Delay readiness until ready	Prevents traffic to bad pods
Add BackoffLimit in Jobs	Stops endless retries

🧪 Incident Snapshot

Time	Event
11:00 AM	5 microservices stuck in CrashLoop
11:02 AM	Env var typo in secret (DB_HOST)
11:05 AM	Secret fixed → pods stable again

☑ Conclusion: Be Outage-Ready, Not Just Cluster-Aware

Knowing how to kubectl get pods is just the surface.
Running a Kubernetes cluster in production is about knowing what breaks — and how to fix it fast.

These 8 outage scenarios represent the **most common real-world failures** DevOps engineers face:

- When your **app is healthy but unreachable**
- When **Pods crash endlessly** for simple reasons
- When **etcd corruption** locks the whole cluster
- When **DNS silently breaks microservice communication**
- When **PVCs prevent pods from ever starting**
- When **Ingress or LoadBalancers misroute live traffic**
- When your **API server collapses under load**
- When the **control plane itself stops responding**

🧠 Want to Stand Out in DevOps Interviews?

❌ Don't just say: *"I deployed apps to Kubernetes."*
☑ Say: *"I simulated production outages in my home lab and recovered from them—DNS failures, LoadBalancer issues, PVC crashes, API slowdowns, and etcd loss."*

That's what real DevOps confidence sounds like.

💡 Your Next Step:

🔁 Practice these scenarios in your home lab
▧ Document how you fixed them
🧪 Build failure-resilient clusters
🎯 Turn these experiences into **high-impact stories** in interviews and on your resume