⚠ INCIDENT REPORT

⚠ The Night Our Pods Went Rogue

Lessons in Kubernetes Reliability

```
$ kubectl get pods | grep CrashLoopBackOff
video-transcoder-6f9d4c7b8d-2j5nx 0/1 CrashLoopBackOff 7 12m
```

By Abhinash Kumar Dubey | DevOps Engineer | Founder @ Kloud-Scaler

Table of Contents

01 Midnight Crisis

02 Kubernetes: The Battlefield

03 Failure States Exposed

04 Anatomy of the Failure

05 CrashLoopBackOff Unpacked

06 Pending, OOMKilled, & ImagePullBackOff

07 The Debugging Process

08 Smarter Probes: Practical Adjustments

09 Resource Requests & Scaling

10 Building Pod Resilience

11 Key Lessons Learned

12 About the Author & Discussion

```
$ kubectl get events --sort-by='.lastTimestamp'
```

01

Midnight Crisis: PagerDuty Strikes

2 AM, PagerDuty & Slack explode. The critical video transcoding service is down. User uploads failing. Business at risk. The drama begins.

CRITICAL ALERT

⚠ Service: video-transcoder
⚠ Status: 17 pods in failure state
⚠ Impact: 100% user uploads failing

```
$ kubectl get pods -n production
video-transcoder-main-1 0/1 CrashLoopBackOff 12 27m
video-transcoder-main-2 0/1 CrashLoopBackOff 9 25m
video-transcoder-proc-1 0/1 OOMKilled 3 24m
video-transcoder-proc-2 0/1 Pending 0 22m
uploads-api-1 1/1 Running 0 1h
```

First wave of panic: Kubernetes wasn't self-healing as expected

Support tickets: 147↑

02

Kubernetes: The Battlefield

Kubernetes was meant to self-heal—but pods are rapidly entering CrashLoopBackOff, Pending, OOMKilled. Cluster is out of control. Fear creeps in.

Kubernetes Cluster Status

🟥 CrashLoopBackOff 🟨 Pending 🟧 OOMKilled 🟦 Running

Rapid Pod Failures

```
$ watch -n 1 kubectl get pods -n production

Every 1.0s: kubectl get pods -n production

NAME READY STATUS RESTARTS AGE
transcoder-7cf74d57f-2dxvp 0/1 CrashLoopBackOff 6 12m
transcoder-7cf74d57f-3p9x1 0/1 OOMKilled 4 17m
transcoder-7cf74d57f-8j2ht 0/1 Pending 0 5m
transcoder-7cf74d57f-f9xk3 0/1 CrashLoopBackOff 7 15m
```

"For the first time, I saw Kubernetes not as a savior, but as a battlefield."

Self-healing: Failed

Failure States Exposed

Common failure states observed during the midnight incident:

CrashLoopBackOff

Pods repeatedly crashing and restarting due to misconfigured liveness probes and missing environment variables.

Pending

Pods stuck in scheduling limbo because no node had sufficient resources available to run them.

OOMKilled

Transcoding jobs consuming excessive memory with no limits set, forcefully terminated by the Linux kernel's OOM killer.

```
$ kubectl get pods -n production | grep -v Running
video-transcoder-main-1 0/1 CrashLoopBackOff 12 27m
video-transcoder-proc-2 0/1 Pending 0 22m
video-transcoder-proc-1 0/1 OOMKilled 3 24m
video-transcoder-proc-3 0/1 ImagePullBackOff 2 18m
video-transcoder-aux-1 0/1 Evicted 0 15m
```

ImagePullBackOff

Pods failing to start because they couldn't pull container images due to incorrect tags or missing registry credentials.

Evicted

Pods forcefully removed from nodes when resource exhaustion threatened node stability.

ℹ Each failure state tells its own story. Each demands a different solution.

Total failures: 17 pods

Anatomy of the Failure

Breaking down the root causes behind our Kubernetes chaos:

OOMKilled everywhere

No CPU/memory limits defined for containers. Pods consuming unlimited resources until killed by kernel.

Readiness probe failures

Probes checking too early, killing pods before they could properly initialize and warm up.

Liveness probe misfires

Incorrectly configured thresholds caused busy pods to be restarted unnecessarily during heavy load.

Pending pods

No resources available on cluster nodes. Scheduler couldn't place new pods.

Node dependency (SPOF)

Multiple replicas placed on same node. No anti-affinity rules to distribute workload.

```
$ kubectl describe pod video-transcoder-main-1 -n production
...
Last State: Terminated
Reason: OOMKilled
Exit Code: 137
Limits: none
Requests: none
```

> Kubernetes wasn't broken—it was brutally honest, showing exactly what we'd ignored.

⚠ 07

CrashLoopBackOff, OOMKilled, Pending Explained

Understanding pod failure states is critical for diagnosing and fixing Kubernetes issues

🔄 CrashLoopBackOff

— Pods repeatedly fail and restart
— Container exits with non-zero status
— Restart delay increases exponentially

Common causes:
- Config/dependency errors
- Missing environment variables
- Misconfigured liveness probes

💣 OOMKilled

— Pod exceeds memory limits
— Killed by Linux kernel OOM killer
— Exit code 137 signature

Common causes:
- No memory limits configured
- Memory leaks in application
- Insufficient resources allocated

🕒 Pending

— Pod scheduled but not running
— Waiting for resources to be available
— Could be stuck indefinitely

Common causes:
- Node resource constraints
- Node selector/affinity issues
- PVC binding failures

```
$ kubectl describe pod video-transcoder-main-1

Status: Failed
Last State: Terminated
Reason: CrashLoopBackOff
Exit Code: 1
Restart Count: 12
Started: Sat, 06 Sep 2025 02:14:38
Finished: Sat, 06 Sep 2025 02:15:02
Events:
Warning: BackOff: Back-off restarting failed container
```

💡 Each failure state requires a different debugging approach and solution

06

The Debugging Process: Turning Chaos into Clarity

When faced with multiple failing pods, our visibility-first approach revealed the underlying issues and guided our recovery.

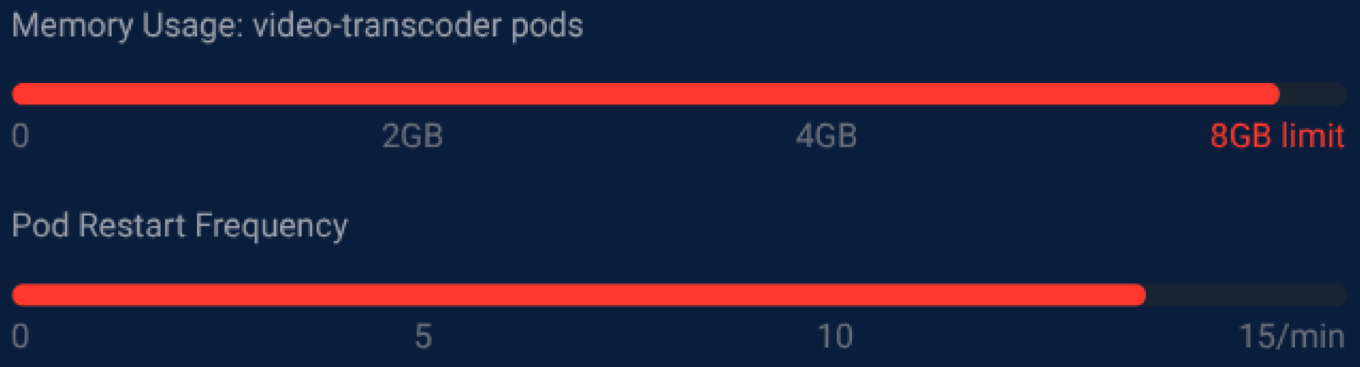
Kubectl Diagnosis

```
$ kubectl describe pod video-transcoder-main-1
Status: Failed
Last State: Terminated
Reason: OOMKilled
Exit Code: 137
Restart Count: 12
Events:
Warning FailedCreatePodSandBox 4m
Normal Pulling 3m image "transcoder:v2.1"
Warning BackOff 2m Back-off restarting
```

Container Logs

```
$ kubectl logs video-transcoder-main-1
[INFO] Initializing transcoder...
[INFO] Loading configuration
[ERROR] Missing env: STORAGE_BUCKET
[ERROR] Transcode failed: Out of memory
```

Grafana Dashboards



❗ Memory spikes correlate with restart events

Cluster-wide Issues

⚠ Node resource constraints detected

- Pods competing for limited memory
- Node CPU throttling observed
- Scheduler unable to place new pods
- Multiple services affected by resource contention

Observability revealed the full story: system-wide resource contention

Time to resolution: -37min

09

Smarter Probes: Adjust & Survive

Fine-tuning Kubernetes probes was critical to prevent false restarts and give pods proper time to initialize and stabilize.

Startup Probe

Allows application time to fully initialize before other probes kick in. Crucial for heavy transcoding containers.

```
✓ Prevents premature restarts
✓ Custom grace period: 90 seconds
✓ Ideal for memory-intensive services
```

Liveness Probe

Detects when pods are unhealthy and triggers restarts, but needs careful threshold tuning.

```
✓ Increased failure threshold: 5
✓ Period seconds: 15
✓ Timeout: 10 seconds
```

Readiness Probe

Controls traffic flow to pods, ensuring only stable pods receive requests.

```
# Transcoder pod configuration with proper probes
startupProbe:
  httpGet:
    path: /healthz
    port: 8080
  failureThreshold: 30
  periodSeconds: 3

livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  failureThreshold: 5
  periodSeconds: 15

readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
```

Lesson: Probes can save you—or kill you.

Pod restart reduction: 92%↓

Resource Requests & Scaling Resilience

Adding proper resource controls and automatic scaling was the key to stability. Without bounds, pods were chaos factories.

CPU/Memory Constraints

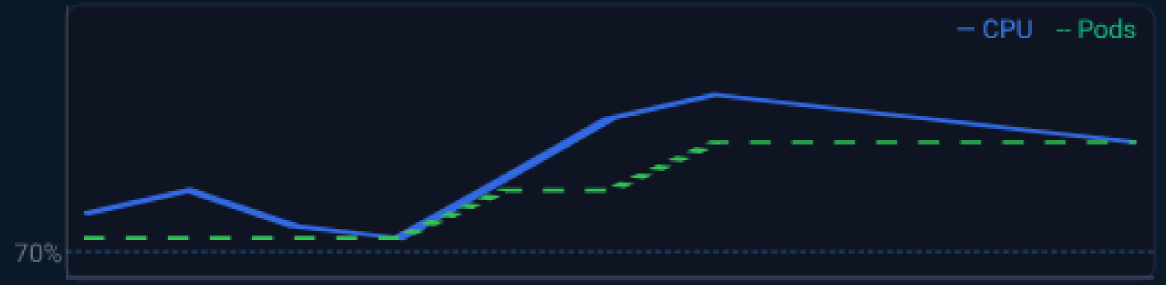
```yaml
# Before: Unbounded resources
apiVersion: apps/v1
kind: Deployment
metadata:
  name: video-transcoder
spec:
  template:
    spec:
      # No resource requests/limits!
```

```yaml
# After: Bounded resources
spec:
  containers:
  - name: transcoder
    resources:
      requests:
        memory: "2Gi"
        cpu: "500m"
      limits:
        memory: "4Gi"
        cpu: "1000m"
```

Automatic Scaling

```yaml
# Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: video-transcoder-hpa
spec:
  minReplicas: 3
  maxReplicas: 10
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: video-transcoder
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

CPU Usage vs. Pod Count (after fix)



RESULTS

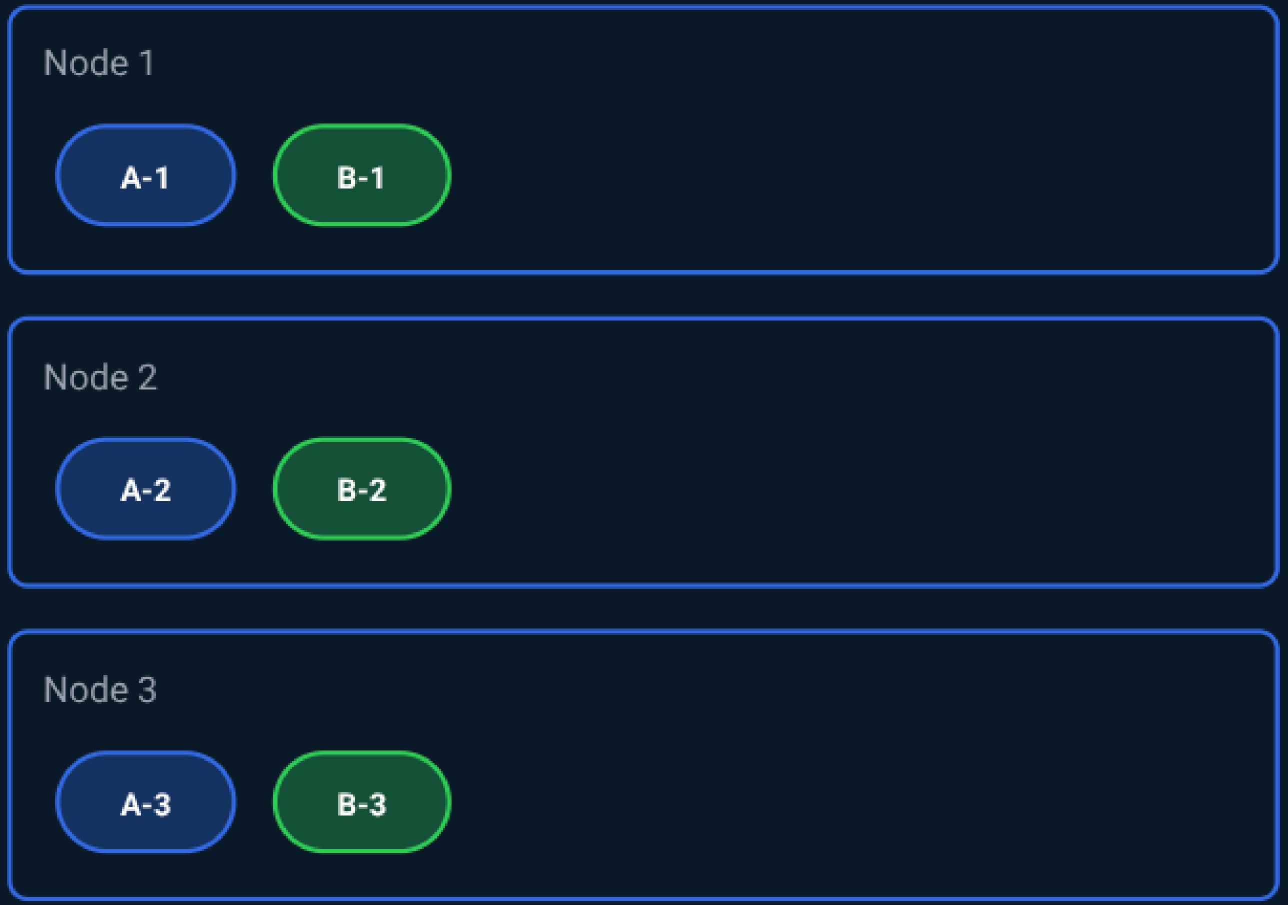
✗ OOMKills: ~~42/day~~ → 0/day
✗ Crashes: ~~29/day~~ → 0/day
✓ Uptime: ~~76%~~ → 99.97%
✓ User experience: Restored

Lesson: Resource limits aren't optional - they're your safety net.

Building Real Reliability

Beyond basic fixes, we implemented architectural patterns to ensure true resilience in our Kubernetes cluster.

Pod Distribution Strategy



Node 1

A-1 B-1

Node 2

A-2 B-2

Node 3

A-3 B-3

✓ Anti-affinity spreads pods across nodes

Resilience Configurations

```yaml
# Pod Anti-Affinity - Spread replicas across nodes
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - topologyKey: "kubernetes.io/hostname"

# PodDisruptionBudget - Maintain minimum availability
apiVersion: policy/v1
kind: PodDisruptionBudget
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: video-transcoder
```

RESILIENCE IMPROVEMENTS

Pod Anti-Affinity
Prevents SPOF by distributing workloads

PodDisruptionBudget
Guarantees service availability during node maintenance

Node Monitoring
Prevents silent evictions with early warnings

Key Lessons & Takeaways

Key Learnings

Pods will fail:
Be ready for CrashLoopBackOff, Pending, OOMKilled—design for recovery.

Probe configuration is critical:
Tune startup, readiness, and liveness probes properly.

Resource requests/limits are non-negotiable:
They ensure proper scheduling and prevent OOM kills.

Scaling equals resilience:
HPA/VPA provide dynamic protection against load spikes.

Reliability requires design:
Anti-affinity + PDB create true fault tolerance.

About the Author

AD

Abhinash Kumar Dubey
DevOps Engineer | Cloud & Kubernetes Enthusiast
Founder @ Kloud-Scaler

techlingo.in abhinash-dubey.techlingo.in

Discussion Question

What's the worst pod failure you've faced in Kubernetes, and how did you solve it? Drop your story in the comments—I'd love to hear how others have battled the same challenges.

"Kubernetes doesn't prevent failures—it helps you survive them, if you respect the system and set it up right."