Scenario #1: Zombie Pods Causing NodeDrain to Hang

- Category: Cluster Management

- Environment: K8s v1.23, On-prem bare metal, Systemd cgroups
  Scenario

- Summary: Node drain stuck indefinitely due to unresponsive terminating pod.

- What Happened:A pod with a custom finalizer never completed termination, blocking kubectl drain. Even after the pod was marked for deletion, the API server kept waiting because the finalizer wasn't removed.

- Diagnosis Steps:

  • Checked kubectl get pods --all-namespaces -o wide to find lingering pods.
  • Found pod stuck in Terminating state for over 20 minutes.
  • Used kubectl describe pod <pod> to identify the presence of a custom finalizer.
  • Investigated controller logs managing the finalizer – the controller had crashed.

  Root Cause: Finalizer logic was never executed because its controller was down, leaving the pod undeletable.

- Fix/Workaround:

```
kubectl patch pod <pod-name> -p '{"metadata":{"finalizers":[]}}' --type=merge
```

- Lessons Learned:
  Finalizers should have timeout or fail-safe logic.

- How to Avoid:

- Avoid finalizers unless absolutely necessary.
- Add monitoring for stuck Terminating pods.
- Implement retry/timeout logic in finalizer controllers.

Scenario #2: API Server Crash Due to Excessive CRD Writes

- Category: Cluster Management

- Environment: K8s v1.24, GKE, heavy use of custom controllers
  Scenario

- Summary: API server crashed due to flooding by a malfunctioning controller creating too many custom resources.

- What Happened:A bug in a controller created thousands of Custom Resources (CRs) in a tight reconciliation loop. Etcd was flooded, leading to slow writes, and the API server eventually became non-responsive.

- Diagnosis Steps:

- API latency increased, leading to 504 Gateway Timeout errors in kubectl.
- Used kubectl get crds | wc -l to list all CRs.
- Analyzed controller logs – found infinite reconcile on a specific CR type.
- etcd disk I/O was maxed.

Root Cause: Bad logic in reconcile loop: create was always called regardless of the state, creating resource floods.

- Fix/Workaround:

  • Scaled the controller to 0 replicas.

  • Manually deleted thousands of stale CRs using batch deletion.

- Lessons Learned:

  Always test reconcile logic in a sandboxed cluster.

- How to Avoid:

  • Implement create/update guards in reconciliation.

  • Add Prometheus alert for high CR count.

Scenario #3: Node Not Rejoining After Reboot

- Category: Cluster Management

- Environment: K8s v1.21, Self-managed cluster, Static nodes

  Scenario

- Summary: A rebooted node failed to rejoin the cluster due to kubelet identity mismatch.

- What Happened:After a kernel upgrade and reboot, a node didn’t appear in kubectl get nodes. The kubelet logs showed registration issues.

- Diagnosis Steps:

  • Checked system logs and kubelet logs.

  • Noticed --hostname-override didn't match the node name registered earlier.

  • kubectl get nodes -o wide showed old hostname; new one mismatched due to DHCP/hostname change.

  Root Cause: Kubelet registered with a hostname that no longer matched its node identity in the cluster.

- Fix/Workaround:

  • Re-joined the node using correct --hostname-override.

  • Cleaned up stale node entry from the cluster.

- Lessons Learned:

  Node identity must remain consistent across reboots.

- How to Avoid:

  • Set static hostnames and IPs.

  • Use consistent cloud-init or kubeadm configuration.

Scenario #4: Etcd Disk Full Causing API Server Timeout

- Category: Cluster Management

- Environment: K8s v1.25, Bare-metal cluster

Scenario

- Summary: etcd ran out of disk space, making API server unresponsive.

- What Happened:The cluster started failing API requests. Etcd logs showed disk space errors, and API server logs showed failed storage operations.

- Diagnosis Steps:

• Used df -h on etcd nodes — confirmed disk full.

• Reviewed /var/lib/etcd – excessive WAL and snapshot files.

• Used etcdctl to assess DB size.

Root Cause: Lack of compaction and snapshotting caused disk to fill up with historical revisions and WALs.

- Fix/Workaround:

```
etcdctl compact <rev>

etcdctl defrag
```

• Cleaned logs, snapshots, and increased disk space temporarily.

- Lessons Learned:

etcd requires periodic maintenance.

- How to Avoid:

• Enable automatic compaction.

• Monitor disk space usage of etcd volumes.

Scenario #5: Misconfigured Taints Blocking Pod Scheduling

- Category: Cluster Management

- Environment: K8s v1.26, Multi-tenant cluster

Scenario

- Summary: Critical workloads weren't getting scheduled due to incorrect node taints.

- What Happened:A user added taints (NoSchedule) to all nodes to isolate their app, but forgot to include tolerations in workloads. Other apps stopped working.

- Diagnosis Steps:

• Pods stuck in Pending state.

• Used kubectl describe pod <pod> – reason: no nodes match tolerations.

• Inspected node taints via kubectl describe node.

Root Cause: Lack of required tolerations on most workloads.

- Fix/Workaround:

• Removed the inappropriate taints.

• Re-scheduled workloads.

- Lessons Learned:

Node taints must be reviewed cluster-wide.

- How to Avoid:

• Educate teams on node taints and tolerations.

• Restrict RBAC for node mutation.

Scenario #6: Kubelet DiskPressure Loop on Large Image Pulls

- Category: Cluster Management

- Environment: K8s v1.22, EKS

Scenario

- Summary: Continuous pod evictions caused by DiskPressure due to image bloating.

- What Happened:A new container image with many layers was deployed. Node's disk filled up, triggering kubelet's DiskPressure condition. Evicted pods created a loop.

- Diagnosis Steps:

• Checked node conditions: kubectl describe node showed DiskPressure: True.

• Monitored image cache with crictl images.

• Node /var/lib/containerd usage exceeded threshold.

Root Cause: Excessive layering in container image and high pull churn caused disk exhaustion.

- Fix/Workaround:

• Rebuilt image using multistage builds and removed unused layers.

• Increased ephemeral disk space temporarily.

- Lessons Learned:

Container image size directly affects node stability.

- How to Avoid:

• Set resource requests/limits appropriately.

• Use image scanning to reject bloated images.

Scenario #7: Node Goes NotReady Due to Clock Skew

- Category: Cluster Management

- Environment: K8s v1.20, On-prem

Scenario

- Summary: One node dropped from the cluster due to TLS errors from time skew.

- What Happened:TLS handshakes between the API server and a node started failing. Node became NotReady. Investigation showed NTP daemon was down.

- Diagnosis Steps:

  • Checked logs for TLS errors: “certificate expired or not yet valid”.

  • Used timedatectl to check drift – node was 45s behind.

  • NTP service was inactive.

  Root Cause: Large clock skew between node and control plane led to invalid TLS sessions.

- Fix/Workaround:

  • Restarted NTP sync.

  • Restarted kubelet after sync.

- Lessons Learned:

  Clock sync is critical in TLS-based distributed systems.

- How to Avoid:

  • Use chronyd or systemd-timesyncd.

  • Monitor clock skew across nodes.

Scenario #8: API Server High Latency Due to Event Flooding

- Category: Cluster Management

- Environment: K8s v1.23, Azure AKS

  Scenario

- Summary: An app spamming Kubernetes events slowed down the entire API server.

- What Happened:A custom controller logged frequent events (~50/second), causing the etcd event store to choke.

- Diagnosis Steps:

  • Prometheus showed spike in event count.

  • kubectl get events --sort-by=.metadata.creationTimestamp showed massive spam.

  • Found misbehaving controller repeating failure events.

  Root Cause: No rate limiting on event creation in controller logic.

- Fix/Workaround:

  • Patched controller to rate-limit record.Eventf.

  • Cleaned old events.

- Lessons Learned:

  Events are not free – they impact etcd/API server.

- How to Avoid:

  • Use deduplicated or summarized event logic.

  • Set API server --event-ttl=1h and --eventRateLimit.

Scenario #9: CoreDNS CrashLoop on Startup

- Category: Cluster Management

- Environment: K8s v1.24, DigitalOcean

  Scenario

- Summary: CoreDNS pods kept crashing due to a misconfigured Corefile.

- What Happened:A team added a custom rewrite rule in the Corefile which had invalid syntax. CoreDNS failed to start.

- Diagnosis Steps:

  • Checked logs: syntax error on startup.

  • Used kubectl describe configmap coredns -n kube-system to inspect.

  • Reproduced issue in test cluster.

  Root Cause: Corefile misconfigured – incorrect directive placement.

- Fix/Workaround:

  • Reverted to backup configmap.

  • Restarted CoreDNS.

- Lessons Learned:

DNS misconfigurations can cascade quickly.

- How to Avoid:

• Use a CoreDNS validator before applying config.

• Maintain versioned backups of Corefile.

Scenario #10: Control Plane Unavailable After Flannel Misconfiguration

- Category: Cluster Management

- Environment: K8s v1.18, On-prem, Flannel CNI

Scenario

- Summary: Misaligned pod CIDRs caused overlay misrouting and API server failure.

- What Happened:A new node was added with a different pod CIDR than what Flannel expected. This broke pod-to-pod and node-to-control-plane communication.

- Diagnosis Steps:

• kubectl timed out from nodes.

• Logs showed dropped traffic in iptables.

• Compared --pod-cidr in kubelet and Flannel config.

Root Cause: Pod CIDRs weren’t consistent across node and Flannel.

- Fix/Workaround:

 • Reconfigured node with proper CIDR range.

 • Flushed iptables and restarted Flannel.

- Lessons Learned:

 CNI requires strict configuration consistency.

- How to Avoid:

 • Enforce CIDR policy via admission control.

 • Validate podCIDR ranges before adding new nodes.

Scenario #11: kube-proxy IPTables Rules Overlap Breaking Networking

- Category: Cluster Management

- Environment: K8s v1.22, On-prem with kube-proxy in IPTables mode

 Scenario

- Summary: Services became unreachable due to overlapping custom IPTables rules with kube-proxy rules.

- What Happened:A system admin added custom IPTables NAT rules for external routing, which inadvertently modified the same chains managed by kube-proxy.

- Diagnosis Steps:

• DNS and service access failing intermittently.

• Ran iptables-save | grep KUBE- – found modified chains.

• Checked kube-proxy logs: warnings about rule insert failures.

Root Cause: Manual IPTables rules conflicted with KUBE-SERVICES chains, causing rule precedence issues.

- Fix/Workaround:

• Flushed custom rules and reloaded kube-proxy.

bash

CopyEdit

```
iptables -F; systemctl restart kube-proxy
```

- Lessons Learned:

Never mix manual IPTables rules with kube-proxy-managed chains.

- How to Avoid:

• Use separate IPTables chains or policy routing.

• Document any node-level firewall rules clearly.

Scenario #12: Stuck CSR Requests Blocking New Node Joins

- Category: Cluster Management

- Environment: K8s v1.20, kubeadm cluster

Scenario

- Summary: New nodes couldn't join due to a backlog of unapproved CSRs.

- What Happened:A spike in expired certificate renewals caused hundreds of CSRs to queue, none of which were being auto-approved. New nodes waited indefinitely.

- Diagnosis Steps:

• Ran kubectl get csr – saw >500 pending requests.

• New nodes stuck at kubelet: "waiting for server signing".

• Approval controller was disabled due to misconfiguration.

Root Cause: Auto-approval for CSRs was turned off during a security patch, but not re-enabled.

- Fix/Workaround:

bash

CopyEdit

```
kubectl certificate approve <csr-name>
```

• Re-enabled the CSR approver controller.

- Lessons Learned:

CSR management is critical for kubelet-node communication.

- How to Avoid:

• Monitor pending CSRs.

• Don’t disable kube-controller-manager flags like --cluster-signing-cert-file.

Scenario #13: Failed Cluster Upgrade Due to Unready Static Pods

- Category: Cluster Management

- Environment: K8s v1.21 → v1.23 upgrade, kubeadm

Scenario

- Summary: Upgrade failed when static control plane pods weren’t ready due to invalid manifests.

- What Happened:During upgrade, etcd didn’t come up because its pod manifest had a typo. Kubelet never started etcd, causing control plane install to hang.

- Diagnosis Steps:

• Checked /etc/kubernetes/manifests/etcd.yaml for errors.

• Used journalctl -u kubelet to see static pod startup errors.

• Verified pod not running via crictl ps.

Root Cause: Human error editing the static pod manifest – invalid volumeMount path.

- Fix/Workaround:

• Fixed manifest.

• Restarted kubelet to load corrected pod.

- Lessons Learned:

  Static pods need strict validation.

- How to Avoid:

  • Use YAML linter on static manifests.

  • Backup manifests before upgrade.

Scenario #14: Uncontrolled Logs Filled Disk on All Nodes

- Category: Cluster Management

- Environment: K8s v1.24, AWS EKS, containerd

  Scenario

- Summary: Application pods generated excessive logs, filling up node /var/log.

- What Happened:A debug flag was accidentally enabled in a backend pod, logging hundreds of lines/sec. The journald and container logs filled up all disk space.

- Diagnosis Steps:

  • df -h showed /var/log full.

  • Checked /var/log/containers/ – massive logs for one pod.

  • Used kubectl logs to confirm excessive output.

Root Cause: A log level misconfiguration caused explosive growth in logs.

- Fix/Workaround:

  • Rotated and truncated logs.
  • Restarted container runtime after cleanup.
  • Disabled debug logging.

- Lessons Learned:
  Logging should be controlled and bounded.

- How to Avoid:

  • Set log rotation policies for container runtimes.
  • Enforce sane log levels via CI/CD validation.

Scenario #15: Node Drain Fails Due to PodDisruptionBudget Deadlock

- Category: Cluster Management

- Environment: K8s v1.21, production cluster with HPA and PDB
  Scenario

- Summary: kubectl drain never completed because PDBs blocked eviction.

- What Happened:A deployment had minAvailable: 2 in PDB, but only 2 pods were running. Node drain couldn’t evict either pod without violating PDB.

- Diagnosis Steps:

  • Ran kubectl describe pdb <name> – saw AllowedDisruptions: 0.

  • Checked deployment and replica count.

  • Tried drain – stuck on pod eviction for 10+ minutes.

  Root Cause: PDB guarantees clashed with under-scaled deployment.

- Fix/Workaround:

  • Temporarily edited PDB to reduce minAvailable.

  • Scaled up replicas before drain.

- Lessons Learned:

  PDBs require careful coordination with replica count.

- How to Avoid:

  • Validate PDBs during deployment scale-downs.

  • Create alerts for PDB blocking evictions.

Scenario #16: CrashLoop of Kube-Controller-Manager on Boot

- Category: Cluster Management

- Environment: K8s v1.23, self-hosted control plane

  Scenario

- Summary: Controller-manager crashed on startup due to outdated admission controller configuration.

- What Happened:After an upgrade, the --enable-admission-plugins flag included a deprecated plugin, causing crash.

- Diagnosis Steps:

  • Checked pod logs in /var/log/pods/.

  • Saw panic error: “unknown admission plugin”.

  • Compared plugin list with K8s documentation.

  Root Cause: Version mismatch between config and actual controller-manager binary.

- Fix/Workaround:

  • Removed the deprecated plugin from startup flags.

  • Restarted pod.

- Lessons Learned:

  Admission plugin deprecations are silent but fatal.

- How to Avoid:

  • Track deprecations in each Kubernetes version.

  • Automate validation of startup flags.

Scenario #17: Inconsistent Cluster State After Partial Backup Restore

- Category: Cluster Management

- Environment: K8s v1.24, Velero-based etcd backup

Scenario

- Summary: A partial etcd restore led to stale object references and broken dependencies.

- What Happened:etcd snapshot was restored, but PVCs and secrets weren't included. Many pods failed to mount or pull secrets.

- Diagnosis Steps:

• Pods failed with "volume not found" and "secret missing".

• kubectl get pvc --all-namespaces returned empty.

• Compared resource counts pre- and post-restore.

Root Cause: Restore did not include volume snapshots or Kubernetes secrets, leading to an incomplete object graph.

- Fix/Workaround:

• Manually recreated PVCs and secrets using backups from another tool.

• Redeployed apps.

- Lessons Learned:

etcd backup is not enough alone.

- How to Avoid:

• Use backup tools that support volume + etcd (e.g., Velero with restic).

• Periodically test full cluster restores.

Scenario #18: kubelet Unable to Pull Images Due to Proxy Misconfig

- Category: Cluster Management

- Environment: K8s v1.25, Corporate proxy network

Scenario

- Summary: Nodes failed to pull images from DockerHub due to incorrect proxy environment configuration.

- What Happened:New kubelet config missed NO_PROXY=10.0.0.0/8,kubernetes.default.svc, causing internal DNS failures and image pull errors.

- Diagnosis Steps:

• kubectl describe pod showed ImagePullBackOff.

• Checked environment variables for kubelet via systemctl show kubelet.

• Verified lack of NO_PROXY.

Root Cause: Proxy config caused kubelet to route internal cluster DNS and registry traffic through the proxy.

- Fix/Workaround:

  • Updated kubelet service file to include proper NO_PROXY.

  • Restarted kubelet.

- Lessons Learned:

  Proxies in K8s require deep planning.

- How to Avoid:

  • Always set NO_PROXY with service CIDRs and cluster domains.

  • Test image pulls with isolated nodes first.

Scenario #19: Multiple Nodes Marked Unreachable Due to Flaky Network Interface

- Category: Cluster Management

- Environment: K8s v1.22, Bare-metal, bonded NICs

  Scenario

- Summary: Flapping interface on switch caused nodes to be marked NotReady intermittently.

- What Happened:A network switch port had flapping issues, leading to periodic loss of node heartbeats.

- Diagnosis Steps:

  • Node status flapped between Ready and NotReady.

  • Checked NIC logs via dmesg and ethtool.

  • Observed link flaps in switch logs.

  Root Cause: Hardware or cable issue causing loss of connectivity.

- Fix/Workaround:

  • Replaced cable and switch port.

  • Set up redundant bonding with failover.

- Lessons Learned:

  Physical layer issues can appear as node flakiness.

- How to Avoid:

  • Monitor NIC link status and configure bonding.

  • Proactively audit switch port health.

Scenario #20: Node Labels Accidentally Overwritten by DaemonSet

- Category: Cluster Management

- Environment: K8s v1.24, DaemonSet-based node config

Scenario

- Summary: A DaemonSet used for node labeling overwrote existing labels used by schedulers.

- What Happened:A platform team deployed a DaemonSet that set node labels like zone=us-east, but it overwrote custom labels like gpu=true.

- Diagnosis Steps:

• Pods no longer scheduled to GPU nodes.

• kubectl get nodes --show-labels showed gpu label missing.

• Checked DaemonSet script – labels were overwritten, not merged.

Root Cause: Label management script used kubectl label node <node> key=value --overwrite, removing other labels.

- Fix/Workaround:

• Restored original labels from backup.

• Updated script to merge labels.

- Lessons Learned:

Node labels are critical for scheduling decisions.

- How to Avoid:

• Use label merging logic (e.g., fetch current labels, then patch).

• Protect key node labels via admission controllers.

Scenario #21: Cluster Autoscaler Continuously Spawning and Deleting Nodes

- Category: Cluster Management

- Environment: K8s v1.24, AWS EKS with Cluster Autoscaler

Scenario

- Summary: The cluster was rapidly scaling up and down, creating instability in workloads.

- What Happened:A misconfigured deployment had a readiness probe that failed intermittently, making pods seem unready. Cluster Autoscaler detected these as unschedulable, triggering new node provisioning. Once the pod appeared healthy again, Autoscaler would scale down.

- Diagnosis Steps:

• Monitored Cluster Autoscaler logs (kubectl -n kube-system logs -l app=cluster-autoscaler).

• Identified repeated scale-up and scale-down messages.

• Traced back to a specific deployment's readiness probe.

Root Cause: Flaky readiness probe created false unschedulable pods.

- Fix/Workaround:

• Fixed the readiness probe to accurately reflect pod health.

• Tuned scale-down-delay-after-add and scale-down-unneeded-time settings.

- Lessons Learned:

Readiness probes directly impact Autoscaler decisions.

- How to Avoid:

• Validate all probes before production deployments.

• Use Autoscaler logging to audit scaling activity.

Scenario #22: Stale Finalizers Preventing Namespace Deletion

- Category: Cluster Management

- Environment: K8s v1.21, self-managed

Scenario

- Summary: A namespace remained in "Terminating" state indefinitely.

- What Happened:The namespace contained resources with finalizers pointing to a deleted controller. Kubernetes waited forever for the finalizer to complete cleanup.

- Diagnosis Steps:

• Ran kubectl get ns <name> -o json – saw dangling finalizers.

• Checked for the corresponding CRD/controller – it was uninstalled.

Root Cause: Finalizers without owning controller cause resource lifecycle deadlocks.

- Fix/Workaround:

• Manually removed finalizers using a patched JSON:

bash

CopyEdit

```
kubectl patch ns <name> -p '{"spec":{"finalizers":[]}}' --type=merge
```

- Lessons Learned:

Always delete CRs before removing the CRD or controller.

- How to Avoid:

• Implement controller cleanup logic.

• Audit finalizers periodically.

Scenario #23: CoreDNS CrashLoop Due to Invalid ConfigMap Update

- Category: Cluster Management

- Environment: K8s v1.23, managed GKE

Scenario

- Summary: CoreDNS stopped resolving names cluster-wide after a config update.

- What Happened:A platform engineer edited the CoreDNS ConfigMap to add a rewrite rule, but introduced a syntax error. The new pods started crashing, and DNS resolution stopped working across the cluster.

- Diagnosis Steps:

  • Ran kubectl logs -n kube-system -l k8s-app=kube-dns – saw config parse errors.

  • Used kubectl describe pod to confirm CrashLoopBackOff.

  • Validated config against CoreDNS docs.

  Root Cause: Invalid configuration line in CoreDNS ConfigMap.

- Fix/Workaround:

  • Rolled back to previous working ConfigMap.

  • Restarted CoreDNS pods to pick up change.

- Lessons Learned:

  ConfigMap changes can instantly affect cluster-wide services.

- How to Avoid:

  • Use coredns -conf <file> locally to validate changes.

  • Test changes in a non-prod namespace before rollout.

Scenario #24: Pod Eviction Storm Due to DiskPressure

- Category: Cluster Management

- Environment: K8s v1.25, self-managed, containerd

  Scenario

- Summary: A sudden spike in image pulls caused all nodes to hit disk pressure, leading to massive pod evictions.

- What Happened:A nightly batch job triggered a container image update across thousands of pods. Pulling these images used all available space in /var/lib/containerd, which led to node condition DiskPressure, forcing eviction of critical workloads.

- Diagnosis Steps:

  • Used kubectl describe node – found DiskPressure=True.

  • Inspected /var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/.

  • Checked image pull logs.

  Root Cause: No image GC and too many simultaneous pulls filled up disk space.

- Fix/Workaround:

  • Pruned unused images.

  • Enabled container runtime garbage collection.

- Lessons Learned:

  DiskPressure can take down entire nodes without warning.

- How to Avoid:

• Set eviction thresholds properly in kubelet.

• Enforce rolling update limits (maxUnavailable).

Scenario #25: Orphaned PVs Causing Unscheduled Pods

- Category: Cluster Management

- Environment: K8s v1.20, CSI storage on vSphere

Scenario

- Summary: PVCs were stuck in Pending state due to existing orphaned PVs in Released state.

- What Happened:After pod deletion, PVs went into Released state but were never cleaned up due to missing ReclaimPolicy logic. When new PVCs requested the same storage class, provisioning failed.

- Diagnosis Steps:

• Ran kubectl get pvc – saw Pending PVCs.

• kubectl get pv – old PVs stuck in Released.

• CSI driver logs showed volume claim conflicts.

Root Cause: ReclaimPolicy set to Retain and no manual cleanup.

- Fix/Workaround:

• Manually deleted orphaned PVs.

• Changed ReclaimPolicy to Delete for similar volumes.

- Lessons Learned:

PV lifecycle must be actively monitored.

- How to Avoid:

• Add cleanup logic in storage lifecycle.

• Implement PV alerts based on state.

Scenario #26: Taints and Tolerations Mismatch Prevented Workload Scheduling

- Category: Cluster Management

- Environment: K8s v1.22, managed AKS

Scenario

- Summary: Workloads failed to schedule on new nodes that had a taint the workloads didn’t tolerate.

- What Happened:Platform team added a new node pool with node-role.kubernetes.io/gpu:NoSchedule, but forgot to add tolerations to GPU workloads.

- Diagnosis Steps:

• kubectl describe pod – showed reason: “0/3 nodes are available: node(s) had taints”.

• Checked node taints via kubectl get nodes -o json.

Root Cause: Taints on new node pool weren't matched by tolerations in pods.

- Fix/Workaround:

• Added proper tolerations to workloads:

yaml

CopyEdit

```
tolerations:

- key: "node-role.kubernetes.io/gpu"
  operator: "Exists"
  effect: "NoSchedule"
```

- Lessons Learned:

Node taints should be coordinated with scheduling policies.

- How to Avoid:

• Use preset toleration templates in CI/CD pipelines.

• Test new node pools with dummy workloads.

Scenario #27: Node Bootstrap Failure Due to Unavailable Container Registry

- Category: Cluster Management

- Environment: K8s v1.21, on-prem, private registry

Scenario

- Summary: New nodes failed to join the cluster due to container runtime timeout when pulling base images.

- What Happened:The internal Docker registry was down during node provisioning, so containerd couldn't pull pauseand CNI images. Nodes stayed in NotReady state.

- Diagnosis Steps:

• journalctl -u containerd – repeated image pull failures.

• Node conditions showed ContainerRuntimeNotReady.

Root Cause: Bootstrap process relies on image pulls from unavailable registry.

- Fix/Workaround:

• Brought internal registry back online.

• Pre-pulled pause/CNI images to node image templates.

- Lessons Learned:

Registry availability is a bootstrap dependency.

- How to Avoid:

• Preload all essential images into AMI/base image.

• Monitor registry uptime independently.

Scenario #28: kubelet Fails to Start Due to Expired TLS Certs

- Category: Cluster Management

- Environment: K8s v1.19, kubeadm cluster

Scenario

- Summary: Several nodes went NotReady after reboot due to kubelet failing to start with expired client certs.

- What Happened:Kubelet uses a client certificate for authentication with the API server. These are typically auto-rotated, but the nodes were offline when the rotation was due.

- Diagnosis Steps:

• journalctl -u kubelet – cert expired error.

• /var/lib/kubelet/pki/kubelet-client-current.pem – expired date.

Root Cause: Kubelet cert rotation missed due to node downtime.

- Fix/Workaround:

• Regenerated kubelet certs using kubeadm.

bash

CopyEdit

```
kubeadm certs renew all
```

- Lessons Learned:

Cert rotation has a dependency on uptime.

- How to Avoid:

• Monitor cert expiry proactively.

• Rotate certs manually before planned outages.

Scenario #29: kube-scheduler Crash Due to Invalid Leader Election Config

- Category: Cluster Management

- Environment: K8s v1.24, custom scheduler deployment

Scenario

- Summary: kube-scheduler pod failed with panic due to misconfigured leader election flags.

- What Happened:An override in the Helm chart introduced an invalid leader election namespace, causing the scheduler to panic and crash on startup.

- Diagnosis Steps:

• Pod logs showed panic: cannot create leader election record.

• Checked Helm values – found wrong namespace name.

Root Cause: Namespace specified for leader election did not exist.

- Fix/Workaround:

  • Created the missing namespace.

  • Restarted the scheduler pod.

- Lessons Learned:

  Leader election is sensitive to namespace scoping.

- How to Avoid:

  • Use default kube-system unless explicitly scoped.

  • Validate all scheduler configs with CI linting.

Scenario #30: Cluster DNS Resolution Broken After Calico CNI Update

- Category: Cluster Management

- Environment: K8s v1.23, self-hosted Calico

  Scenario

- Summary: DNS resolution broke after Calico CNI update due to iptables policy drop changes.

- What Happened:New version of Calico enforced stricter iptables drop policies, blocking traffic from CoreDNS to pods.

- Diagnosis Steps:

• DNS requests timed out.

• Packet capture showed ICMP unreachable from pods to CoreDNS.

• Checked Calico policy and iptables rules.

Root Cause: Calico’s default deny policy applied to kube-dns traffic.

- Fix/Workaround:

• Added explicit Calico policy allowing kube-dns to pod traffic.

```
yaml:
egress:

- action: Allow
  destination:
  selector: "k8s-app == 'kube-dns'"
```

- Lessons Learned:

CNI policy changes can impact DNS without warning.

- How to Avoid:

• Review and test all network policy upgrades in staging.

• Use canary upgrade strategy for CNI.

Scenario #31: Node Clock Drift Causing Authentication Failures

- Category: Cluster Management

- Environment: K8s v1.22, on-prem, kubeadm

Scenario

- Summary: Authentication tokens failed across the cluster due to node clock skew.

- What Happened:Token-based authentication failed for all workloads and kubectl access due to time drift between worker nodes and the API server.

- Diagnosis Steps:

  • Ran kubectl logs and found expired token errors.

  • Checked node time using date on each node – found significant drift.

  • Verified NTP daemon status – not running.

Root Cause: NTP daemon disabled on worker nodes.

- Fix/Workaround:

  • Re-enabled and restarted NTP on all nodes.

  • Synchronized system clocks manually.

- Lessons Learned:

Time synchronization is critical for certificate and token-based auth.

- How to Avoid:

• Ensure NTP or chrony is enabled via bootstrap configuration.

• Monitor time drift via node-exporter.

Scenario #32: Inconsistent Node Labels Causing Scheduling Bugs

- Category: Cluster Management

- Environment: K8s v1.24, multi-zone GKE

Scenario

- Summary: Zone-aware workloads failed to schedule due to missing zone labels on some nodes.

- What Happened:Pods using topologySpreadConstraints for zone balancing failed to find valid nodes because some nodes lacked the topology.kubernetes.io/zone label.

- Diagnosis Steps:

• Pod events showed no matching topology key errors.

• Compared node labels across zones – found inconsistency.

Root Cause: A few nodes were manually added without required zone labels.

- Fix/Workaround:

• Manually patched node labels to restore zone metadata.

- Lessons Learned:

Label uniformity is essential for topology constraints.

- How to Avoid:

• Automate label injection using cloud-init or DaemonSet.

• Add CI checks for required labels on node join.

Scenario #33: API Server Slowdowns from High Watch Connection Count

- Category: Cluster Management

- Environment: K8s v1.23, OpenShift

Scenario

- Summary: API latency rose sharply due to thousands of watch connections from misbehaving clients.

- What Happened:Multiple pods opened persistent watch connections and never closed them, overloading the API server.

- Diagnosis Steps:

• Monitored API metrics /metrics for apiserver_registered_watchers.

• Identified top offenders using connection source IPs.

Root Cause: Custom controller with poor watch logic never closed connections.

- Fix/Workaround:

• Restarted offending pods.

• Updated controller to reuse watches.

- Lessons Learned:

Unbounded watches can exhaust server resources.

- How to Avoid:

• Use client-go with resync periods and connection limits.

• Enable metrics to detect watch leaks early.

Scenario #34: Etcd Disk Full Crashing the Cluster

- Category: Cluster Management

- Environment: K8s v1.21, self-managed with local etcd

Scenario

- Summary: Entire control plane crashed due to etcd disk running out of space.

- What Happened:Continuous writes from custom resources filled the disk where etcd data was stored.

- Diagnosis Steps:

• Observed etcdserver: mvcc: database space exceeded errors.

• Checked disk usage: df -h showed 100% full.

Root Cause: No compaction or defragmentation done on etcd for weeks.

- Fix/Workaround:

• Performed etcd compaction and defragmentation.

• Added disk space temporarily.

- Lessons Learned:

Etcd needs regular maintenance.

- How to Avoid:

• Set up cron jobs or alerts for etcd health.

• Monitor disk usage and trigger auto-compaction.

Scenario #35: ClusterConfigMap Deleted by Accident Bringing Down Addons

- Category: Cluster Management

- Environment: K8s v1.24, Rancher

Scenario

- Summary: A user accidentally deleted the kube-root-ca.crt ConfigMap, which many workloads relied on.

- What Happened:Pods mounting the kube-root-ca.crt ConfigMap failed to start after deletion. DNS, metrics-server, and other system components failed.

- Diagnosis Steps:

  • Pod events showed missing ConfigMap errors.

  • Attempted to remount volumes manually.

  Root Cause: System-critical ConfigMap was deleted without RBAC protections.

- Fix/Workaround:

  • Recreated ConfigMap from backup.

  • Re-deployed affected system workloads.

- Lessons Learned:

  Some ConfigMaps are essential and must be protected.

- How to Avoid:

  • Add RBAC restrictions to system namespaces.

  • Use OPA/Gatekeeper to prevent deletions of protected resources.

Scenario #36: Misconfigured NodeAffinity Excluding All Nodes

- Category: Cluster Management

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: A critical deployment was unschedulable due to strict nodeAffinity rules.

- What Happened:nodeAffinity required a zone that did not exist in the cluster, making all nodes invalid.

- Diagnosis Steps:

• Pod events showed 0/10 nodes available errors.

• Checked spec.affinity section in deployment YAML.

Root Cause: Invalid or overly strict requiredDuringScheduling nodeAffinity.

- Fix/Workaround:

• Updated deployment YAML to reflect actual zones.

• Re-deployed workloads.

- Lessons Learned:

nodeAffinity is strict and should be used carefully.

- How to Avoid:

• Validate node labels before setting affinity.

• Use preferredDuringScheduling for soft constraints.

Scenario #37: Outdated Admission Webhook Blocking All Deployments

- Category: Cluster Management

- Environment: K8s v1.25, self-hosted

  Scenario

- Summary: A stale mutating webhook caused all deployments to fail due to TLS certificate errors.

- What Happened:The admission webhook had expired TLS certs, causing validation errors on all resource creation attempts.

- Diagnosis Steps:

  • Created a dummy pod and observed webhook errors.

  • Checked logs of the webhook pod – found TLS handshake failures.

  Root Cause: Webhook server was down due to expired TLS cert.

- Fix/Workaround:

  • Renewed cert and redeployed webhook.

  • Disabled webhook temporarily for emergency deployments.

- Lessons Learned:

  Webhooks are gatekeepers – they must be monitored.

- How to Avoid:

  • Rotate webhook certs using cert-manager.
  • Alert on webhook downtime or errors.

Scenario #38: API Server Certificate Expiry Blocking Cluster Access

- Category: Cluster Management

- Environment: K8s v1.19, kubeadm

  Scenario

- Summary: After 1 year of uptime, API server certificate expired, blocking access to all components.

- What Happened:Default kubeadm cert rotation didn't occur, leading to expiry of API server and etcd peer certs.

- Diagnosis Steps:

  • kubectl failed with x509: certificate has expired.
  • Checked /etc/kubernetes/pki/apiserver.crt expiry date.

  Root Cause: kubeadm certificates were never rotated or renewed.

- Fix/Workaround:

  • Used kubeadm certs renew all.

• Restarted control plane components.

- Lessons Learned:

Certificates expire silently unless monitored.

- How to Avoid:

• Rotate certs before expiry.

• Monitor /metrics for cert validity.

Scenario #39: CRI Socket Mismatch Preventing kubelet Startup

- Category: Cluster Management

- Environment: K8s v1.22, containerd switch

Scenario

- Summary: kubelet failed to start after switching from Docker to containerd due to incorrect CRI socket path.

- What Happened:The node image had containerd installed, but the kubelet still pointed to the Docker socket.

- Diagnosis Steps:

• Checked kubelet logs for failed to connect to CRI socket.

• Verified config file at /var/lib/kubelet/kubeadm-flags.env.

Root Cause: Wrong --container-runtime-endpoint specified.

- Fix/Workaround:

  • Updated kubelet flags to point to /run/containerd/containerd.sock.
  • Restarted kubelet.

- Lessons Learned:

  CRI migration requires explicit config updates.

- How to Avoid:

  • Use migration scripts or kubeadm migration guides.
  • Validate container runtime on node bootstrap.

Scenario #40: Cluster-Wide Crash Due to Misconfigured Resource Quotas

- Category: Cluster Management

- Environment: K8s v1.24, multi-tenant namespace setup

  Scenario

- Summary: Cluster workloads failed after applying overly strict resource quotas that denied new pod creation.

- What Happened:A new quota was applied with very low CPU/memory limits. All new pods across namespaces failed scheduling.

- Diagnosis Steps:

• Pod events showed failed quota check errors.

• Checked quota via kubectl describe quota in all namespaces.

Root Cause: Misconfigured CPU/memory limits set globally.

- Fix/Workaround:

• Rolled back the quota to previous values.

• Unblocked critical namespaces manually.

- Lessons Learned:

Quota changes should be staged and validated.

- How to Avoid:

• Test new quotas in shadow or dry-run mode.

• Use automated checks before applying quotas.

Scenario #41: Cluster Upgrade Failing Due to CNI Compatibility

- Category: Cluster Management

- Environment: K8s v1.21 to v1.22, custom CNI plugin

Scenario

- Summary: Cluster upgrade failed due to an incompatible version of the CNI plugin.

- What Happened:After upgrading the control plane, CNI plugins failed to work, resulting in no network connectivity between pods.

- Diagnosis Steps:

  • Checked kubelet and container runtime logs – observed CNI errors.

  • Verified CNI plugin version – it was incompatible with K8s v1.22.

  Root Cause: CNI plugin was not upgraded alongside the Kubernetes control plane.

- Fix/Workaround:

  • Upgraded the CNI plugin to the version compatible with K8s v1.22.

  • Restarted affected pods and nodes.

- Lessons Learned:

  Always ensure compatibility between the Kubernetes version and CNI plugin.

- How to Avoid:

  • Follow Kubernetes upgrade documentation and ensure CNI plugins are upgraded.

  • Test in a staging environment before performing production upgrades.

Scenario #42: Failed Pod Security Policy Enforcement Causing Privileged Container Launch

- Category: Cluster Management

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Privileged containers were able to run despite Pod Security Policy enforcement.

- What Happened:A container was able to run as privileged despite a restrictive PodSecurityPolicy being in place.

- Diagnosis Steps:

  • Checked pod events and logs, found no violations of PodSecurityPolicy.

  • Verified PodSecurityPolicy settings and namespace annotations.

Root Cause: PodSecurityPolicy was not enforced due to missing podsecuritypolicy admission controller.

- Fix/Workaround:

  • Enabled the podsecuritypolicy admission controller.

  • Updated the PodSecurityPolicy to restrict privileged containers.

- Lessons Learned:

Admission controllers must be properly configured for security policies to be enforced.

- How to Avoid:

• Double-check admission controller configurations during initial cluster setup.

• Regularly audit security policies and admission controllers.

Scenario #43: Node Pool Scaling Impacting StatefulSets

- Category: Cluster Management

- Environment: K8s v1.24, GKE

Scenario

- Summary: StatefulSet pods were rescheduled across different nodes, breaking persistent volume bindings.

- What Happened:Node pool scaling in GKE triggered a rescheduling of StatefulSet pods, breaking persistent volume claims that were tied to specific nodes.

- Diagnosis Steps:

• Observed failed to bind volume errors.

• Checked StatefulSet configuration for node affinity and volume binding policies.

Root Cause: Lack of proper node affinity or persistent volume binding policies in StatefulSet configuration.

- Fix/Workaround:

• Added proper node affinity rules and volume binding policies to StatefulSet.

• Rescheduled the pods successfully.

- Lessons Learned:

  StatefulSets require careful management of node affinity and persistent volume binding policies.

- How to Avoid:

  • Use pod affinity rules for StatefulSets to ensure proper scheduling and volume binding.

  • Monitor volume binding status when scaling node pools.

Scenario #44: Kubelet Crash Due to Out of Memory (OOM) Errors

- Category: Cluster Management

- Environment: K8s v1.20, bare metal

  Scenario

- Summary: Kubelet crashed after running out of memory due to excessive pod resource usage.

- What Happened:The kubelet on a node crashed after the available memory was exhausted due to pods consuming more memory than allocated.

- Diagnosis Steps:

  • Checked kubelet logs for OOM errors.

• Used kubectl describe node to check resource utilization.

Root Cause: Pod resource requests and limits were not set properly, leading to excessive memory consumption.

- Fix/Workaround:

• Set proper resource requests and limits on pods to prevent memory over-consumption.

• Restarted the kubelet on the affected node.

- Lessons Learned:

Pod resource limits and requests are essential for proper node resource utilization.

- How to Avoid:

• Set reasonable resource requests and limits for all pods.

• Monitor node resource usage to catch resource overuse before it causes crashes.

Scenario #45: DNS Resolution Failure in Multi-Cluster Setup

- Category: Cluster Management

- Environment: K8s v1.23, multi-cluster federation

Scenario

- Summary: DNS resolution failed between two federated clusters due to missing DNS records.

- What Happened:DNS queries failed between two federated clusters, preventing services from accessing each other across clusters.

- Diagnosis Steps:

  • Used kubectl get svc to check DNS records.

  • Identified missing service entries in the DNS server configuration.

  Root Cause: DNS configuration was incomplete, missing records for federated services.

- Fix/Workaround:

  • Added missing DNS records manually.

  • Updated DNS configurations to include service records for all federated clusters.

- Lessons Learned:

  In multi-cluster setups, DNS configuration is critical to service discovery.

- How to Avoid:

  • Automate DNS record creation during multi-cluster federation setup.

  • Regularly audit DNS configurations in multi-cluster environments.

Scenario #46: Insufficient Resource Limits in Autoscaling Setup

- Category: Cluster Management

- Environment: K8s v1.21, GKE with Horizontal Pod Autoscaler (HPA)

Scenario

- Summary: Horizontal Pod Autoscaler did not scale pods up as expected due to insufficient resource limits.

- What Happened:The Horizontal Pod Autoscaler failed to scale the application pods up, even under load, due to insufficient resource limits set on the pods.

- Diagnosis Steps:

• Observed HPA metrics showing no scaling action.

• Checked pod resource requests and limits.

Root Cause: Resource limits were too low for HPA to trigger scaling actions.

- Fix/Workaround:

• Increased resource requests and limits for the affected pods.

• Manually scaled the pods and monitored the autoscaling behavior.

- Lessons Learned:

Proper resource limits are essential for autoscaling to function correctly.

- How to Avoid:

• Set adequate resource requests and limits for workloads managed by HPA.

• Monitor autoscaling events to identify under-scaling issues.

Scenario #47: Control Plane Overload Due to High Audit Log Volume

- Category: Cluster Management

- Environment: K8s v1.22, Azure AKS
  Scenario

- Summary: The control plane became overloaded and slow due to excessive audit log volume.

- What Happened:A misconfigured audit policy led to high volumes of audit logs being generated, overwhelming the control plane.

- Diagnosis Steps:

  • Monitored control plane metrics and found high CPU usage due to audit logs.
  • Reviewed audit policy and found it was logging excessive data.
  Root Cause: Overly broad audit log configuration captured too many events.

- Fix/Workaround:

  • Refined audit policy to log only critical events.
  • Restarted the API server.

- Lessons Learned:
  Audit logging needs to be fine-tuned to prevent overload.

- How to Avoid:

• Regularly review and refine audit logging policies.

• Set alerts for high audit log volumes.

Scenario #48: Resource Fragmentation Causing Cluster Instability

- Category: Cluster Management

- Environment: K8s v1.23, bare metal

Scenario

- Summary: Resource fragmentation due to unbalanced pod distribution led to cluster instability.

- What Happened:Over time, pod distribution became uneven, with some nodes over-committed while others remained underutilized. This caused resource fragmentation, leading to cluster instability.

- Diagnosis Steps:

• Checked node resource utilization and found over-committed nodes with high pod density.

• Examined pod distribution and noticed skewed placement.

Root Cause: Lack of proper pod scheduling and resource management.

- Fix/Workaround:

• Applied pod affinity and anti-affinity rules to achieve balanced scheduling.

• Rescheduled pods manually to redistribute workload.

- Lessons Learned:

Resource management and scheduling rules are crucial for maintaining cluster stability.

- How to Avoid:

• Use affinity and anti-affinity rules to control pod placement.

• Regularly monitor resource utilization and adjust pod placement strategies.

Scenario #49: Failed Cluster Backup Due to Misconfigured Volume Snapshots

- Category: Cluster Management

- Environment: K8s v1.21, AWS EBS

Scenario

- Summary: Cluster backup failed due to a misconfigured volume snapshot driver.

- What Happened:The backup process failed because the EBS volume snapshot driver was misconfigured, resulting in incomplete backups.

- Diagnosis Steps:

• Checked backup logs for error messages related to volume snapshot failures.

• Verified snapshot driver configuration in storage class.

Root Cause: Misconfigured volume snapshot driver prevented proper backups.

- Fix/Workaround:

• Corrected snapshot driver configuration in storage class.

• Ran the backup process again, which completed successfully.

- Lessons Learned:

Backup configuration must be thoroughly checked and tested.

- How to Avoid:

• Automate backup testing and validation in staging environments.

• Regularly verify backup configurations.

Scenario #50: Failed Deployment Due to Image Pulling Issues

- Category: Cluster Management

- Environment: K8s v1.22, custom Docker registry

Scenario

- Summary: Deployment failed due to image pulling issues from a custom Docker registry.

- What Happened:A deployment failed because Kubernetes could not pull images from a custom Docker registry due to misconfigured image pull secrets.

- Diagnosis Steps:

  • Observed ImagePullBackOff errors for the failing pods.

  • Checked image pull secrets and registry configuration.

  Root Cause: Incorrect or missing image pull secrets for accessing the custom registry.

- Fix/Workaround:

  • Corrected the image pull secrets in the deployment YAML.

  • Re-deployed the application.

- Lessons Learned:

  Image pull secrets must be configured properly for private registries.

- How to Avoid:

  • Always verify image pull secrets for private registries.

  • Use Kubernetes secrets management tools for image pull credentials.

Scenario #51: High Latency Due to Inefficient Ingress Controller Configuration

- Category: Cluster Management

- Environment: K8s v1.20, AWS EKS

  Scenario

- Summary: Ingress controller configuration caused high network latency due to inefficient routing rules.

- What Happened:Ingress controller was handling a large number of routes inefficiently, resulting in significant network latency and slow response times for external traffic.

- Diagnosis Steps:

  • Analyzed ingress controller logs for routing delays.

  • Checked ingress resources and discovered unnecessary complex path-based routing rules.

  Root Cause: Inefficient ingress routing rules and too many path-based routes led to slower packet processing.

- Fix/Workaround:

  • Simplified ingress resource definitions and optimized routing rules.

  • Restarted ingress controller to apply changes.

- Lessons Learned:

  Optimizing ingress routing rules is critical for performance, especially in high-traffic environments.

- How to Avoid:

• Regularly review and optimize ingress resources.

• Use a more efficient ingress controller (e.g., NGINX Ingress Controller) for high-volume environments.

Scenario #52: Node Draining Delay During Maintenance

- Category: Cluster Management

- Environment: K8s v1.21, GKE

Scenario

- Summary: Node draining took an unusually long time during maintenance due to unscheduled pod disruption.

- What Happened:During a scheduled node maintenance, draining took longer than expected because pods were not respecting PodDisruptionBudgets.

- Diagnosis Steps:

• Checked kubectl describe for affected pods and identified PodDisruptionBudget violations.

• Observed that some pods had hard constraints on disruption due to storage.

Root Cause: PodDisruptionBudget was too strict, preventing pods from being evicted quickly.

- Fix/Workaround:

• Adjusted PodDisruptionBudget to allow more flexibility for pod evictions.

• Manually evicted the pods to speed up the node draining process.

- Lessons Learned:

PodDisruptionBudgets should be set based on actual disruption tolerance.

- How to Avoid:

• Set reasonable disruption budgets for critical applications.

• Test disruption scenarios during maintenance windows to identify issues.

Scenario #53: Unresponsive Cluster After Large-Scale Deployment

- Category: Cluster Management

- Environment: K8s v1.19, Azure AKS

Scenario

- Summary: Cluster became unresponsive after deploying a large number of pods in a single batch.

- What Happened:The cluster became unresponsive after deploying a batch of 500 pods in a single operation, causing resource exhaustion.

- Diagnosis Steps:

• Checked cluster logs and found that the control plane was overwhelmed with API requests.

• Observed resource limits on the nodes, which were maxed out.

Root Cause: The large-scale deployment exhausted the cluster's available resources, causing a spike in API server load.

- Fix/Workaround:

• Implemented gradual pod deployment using rolling updates instead of a batch deployment.

• Increased the node resource capacity to handle larger loads.

- Lessons Learned:

Gradual deployments and resource planning are necessary when deploying large numbers of pods.

- How to Avoid:

• Use rolling updates or deploy in smaller batches.

• Monitor cluster resources and scale nodes accordingly.

Scenario #54: Failed Node Recovery Due to Corrupt Kubelet Configuration

- Category: Cluster Management

- Environment: K8s v1.23, Bare Metal

Scenario

- Summary: Node failed to recover after being drained due to a corrupt kubelet configuration.

- What Happened:After a node was drained for maintenance, it failed to rejoin the cluster due to a corrupted kubelet configuration file.

- Diagnosis Steps:

  • Checked kubelet logs and identified errors related to configuration loading.

  • Verified kubelet configuration file on the affected node and found corruption.

  Root Cause: A corrupted kubelet configuration prevented the node from starting properly.

- Fix/Workaround:

  • Replaced the corrupted kubelet configuration file with a backup.

  • Restarted the kubelet service and the node successfully rejoined the cluster.

- Lessons Learned:

  Regular backups of critical configuration files like kubelet configs can save time during node recovery.

- How to Avoid:

  • Automate backups of critical configurations.

  • Implement configuration management tools for easier recovery.

Scenario #55: Resource Exhaustion Due to Misconfigured Horizontal Pod Autoscaler

- Category: Cluster Management

- Environment: K8s v1.22, AWS EKS

  Scenario

- Summary: Cluster resources were exhausted due to misconfiguration in the Horizontal Pod Autoscaler (HPA), resulting in excessive pod scaling.

- What Happened:HPA was configured to scale pods based on CPU utilization but had an overly sensitive threshold, causing the application to scale out rapidly and exhaust resources.

- Diagnosis Steps:

  • Analyzed HPA metrics and found excessive scaling actions.

  • Verified CPU utilization metrics and observed that they were consistently above the threshold due to a sudden workload spike.

  Root Cause: HPA was too aggressive in scaling up based on CPU utilization, without considering other metrics like memory usage or custom metrics.

- Fix/Workaround:

  • Adjusted HPA configuration to scale based on a combination of CPU and memory usage.

  • Set more appropriate scaling thresholds.

- Lessons Learned:

Scaling based on a single metric (e.g., CPU) can lead to inefficiency, especially during workload spikes.

- How to Avoid:

• Use multiple metrics for autoscaling (e.g., CPU, memory, and custom metrics).

• Set more conservative scaling thresholds to prevent resource exhaustion.

Scenario #56: Inconsistent Application Behavior After Pod Restart

- Category: Cluster Management

- Environment: K8s v1.20, GKE

Scenario

- Summary: Application behavior became inconsistent after pod restarts due to improper state handling.

- What Happened:After a pod was restarted, the application started behaving unpredictably, with some users experiencing different results from others due to lack of state synchronization.

- Diagnosis Steps:

• Checked pod logs and noticed that state data was being stored in the pod's ephemeral storage.

• Verified that application code did not handle state persistence properly.

Root Cause: The application was not designed to persist state beyond the pod lifecycle, leading to inconsistent behavior after restarts.

- Fix/Workaround:

  • Moved application state to persistent volumes or external databases.

  • Adjusted the application logic to handle state recovery properly after restarts.

- Lessons Learned:

  State should be managed outside of ephemeral storage for applications that require consistency.

- How to Avoid:

  • Use persistent volumes for stateful applications.

  • Implement state synchronization mechanisms where necessary.

Scenario #57: Cluster-wide Service Outage Due to Missing ClusterRoleBinding

- Category: Cluster Management

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Cluster-wide service outage occurred after an automated change removed a critical ClusterRoleBinding.

- What Happened:A misconfigured automation pipeline accidentally removed a ClusterRoleBinding, which was required for certain critical services to function.

- Diagnosis Steps:

  • Analyzed service logs and found permission-related errors.

  • Checked the RBAC configuration and found the missing ClusterRoleBinding.

  Root Cause: Automated pipeline incorrectly removed the ClusterRoleBinding, causing service permissions to be revoked.

- Fix/Workaround:

  • Restored the missing ClusterRoleBinding.

  • Manually verified that affected services were functioning correctly.

- Lessons Learned:

  Automation changes must be reviewed and tested to prevent accidental permission misconfigurations.

- How to Avoid:

  • Use automated tests and checks for RBAC changes.

  • Implement safeguards and approval workflows for automated configuration changes.

Scenario #58: Node Overcommitment Leading to Pod Evictions

- Category: Cluster Management

- Environment: K8s v1.19, Bare Metal

  Scenario

- Summary: Node overcommitment led to pod evictions, causing application downtime.

- What Happened:Due to improper resource requests and limits, the node was overcommitted, which led to the eviction of critical pods.

- Diagnosis Steps:

  • Checked the node's resource utilization and found it was maxed out.

  • Analyzed pod logs to see eviction messages related to resource limits.

  Root Cause: Pods did not have properly set resource requests and limits, leading to resource overcommitment on the node.

- Fix/Workaround:

  • Added appropriate resource requests and limits to the affected pods.

  • Rescheduled the pods to other nodes with available resources.

- Lessons Learned:

  Properly setting resource requests and limits prevents overcommitment and avoids pod evictions.

- How to Avoid:

  • Always set appropriate resource requests and limits for all pods.

  • Use resource quotas and limit ranges to prevent overcommitment.

Scenario #59: Failed Pod Startup Due to Image Pull Policy Misconfiguration

- Category: Cluster Management

- Environment: K8s v1.23, Azure AKS

Scenario

- Summary: Pods failed to start because the image pull policy was misconfigured.

- What Happened:The image pull policy was set to Never, preventing Kubernetes from pulling the required container images from the registry.

- Diagnosis Steps:

• Checked pod events and found image pull errors.

• Verified the image pull policy in the pod specification.

Root Cause: Image pull policy was set to Never, which prevents Kubernetes from pulling images from the registry.

- Fix/Workaround:

• Changed the image pull policy to IfNotPresent or Always in the pod configuration.

• Re-deployed the pods.

- Lessons Learned:

The correct image pull policy is necessary to ensure Kubernetes can pull container images from a registry.

- How to Avoid:

• Double-check the image pull policy in pod specifications before deployment.

• Use Always for images stored in remote registries.

Scenario #60: Excessive Control Plane Resource Usage During Pod Scheduling

- Category: Cluster Management

- Environment: K8s v1.24, AWS EKS

Scenario

- Summary: Control plane resources were excessively utilized during pod scheduling, leading to slow deployments.

- What Happened:Pod scheduling took significantly longer than expected due to excessive resource consumption in the control plane.

- Diagnosis Steps:

• Monitored control plane metrics and found high CPU and memory usage.

• Analyzed scheduler logs to identify resource bottlenecks.

Root Cause: The default scheduler was not optimized for high resource consumption, causing delays in pod scheduling.

- Fix/Workaround:

  • Optimized the scheduler configuration to reduce resource usage.

  • Split large workloads into smaller ones to improve scheduling efficiency.

- Lessons Learned:

  Efficient scheduler configuration is essential for handling large-scale deployments.

- How to Avoid:

  • Optimize scheduler settings for large clusters.

  • Use scheduler features like affinity and anti-affinity to control pod placement.

Scenario #61: Persistent Volume Claim Failure Due to Resource Quota Exceedance

- Category: Cluster Management

- Environment: K8s v1.22, GKE

  Scenario

- Summary: Persistent Volume Claims (PVCs) failed due to exceeding the resource quota for storage in the namespace.

- What Happened:A user attempted to create PVCs that exceeded the available storage quota, leading to failed PVC creation.

- Diagnosis Steps:

  • Checked the namespace resource quotas using kubectl get resourcequotas.

  • Observed that the storage limit had been reached.

  Root Cause: PVCs exceeded the configured storage resource quota in the namespace.

- Fix/Workaround:

  • Increased the storage quota in the namespace.

  • Cleaned up unused PVCs to free up space.

- Lessons Learned:

  Proper resource quota management is critical for ensuring that users cannot overuse resources.

- How to Avoid:

  • Regularly review and adjust resource quotas based on usage patterns.

  • Implement automated alerts for resource quota breaches.

Scenario #62: Failed Pod Rescheduling Due to Node Affinity Misconfiguration

- Category: Cluster Management

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Pods failed to reschedule after a node failure due to improper node affinity rules.

- What Happened:When a node was taken down for maintenance, the pod failed to reschedule due to restrictive node affinity settings.

- Diagnosis Steps:

  • Checked pod events and noticed affinity rule errors preventing the pod from scheduling on other nodes.

  • Analyzed the node affinity configuration in the pod spec.

  Root Cause: Node affinity rules were set too restrictively, preventing the pod from being scheduled on other nodes.

- Fix/Workaround:

  • Adjusted the node affinity rules to be less restrictive.

  • Re-scheduled the pods to available nodes.

- Lessons Learned:

  Affinity rules should be configured to provide sufficient flexibility for pod rescheduling.

- How to Avoid:

  • Set node affinity rules based on availability and workloads.

  • Regularly test affinity and anti-affinity rules during node maintenance windows.

Scenario #63: Intermittent Network Latency Due to Misconfigured CNI Plugin

- Category: Cluster Management

- Environment: K8s v1.24, Azure AKS

  Scenario

- Summary: Network latency issues occurred intermittently due to misconfiguration in the CNI (Container Network Interface) plugin.

- What Happened:Network latency was sporadically high between pods due to improper settings in the CNI plugin.

- Diagnosis Steps:

  • Analyzed network metrics and noticed high latency between pods in different nodes.

  • Checked CNI plugin logs and configuration and found incorrect MTU (Maximum Transmission Unit) settings.

  Root Cause: MTU misconfiguration in the CNI plugin caused packet fragmentation, resulting in network latency.

- Fix/Workaround:

  • Corrected the MTU setting in the CNI configuration to match the network infrastructure.

  • Restarted the CNI plugin and verified network performance.

- Lessons Learned:

Proper CNI configuration is essential to avoid network latency and connectivity issues.

- How to Avoid:

  • Ensure CNI plugin configurations match the underlying network settings.
  • Test network performance after changes to the CNI configuration.

Scenario #64: Excessive Pod Restarts Due to Resource Limits

- Category: Cluster Management

- Environment: K8s v1.19, GKE

  Scenario

- Summary: A pod was restarting frequently due to resource limits being too low, causing the container to be killed.

- What Happened:Pods were being killed and restarted due to the container's resource requests and limits being set too low, causing OOM (Out of Memory) kills.

- Diagnosis Steps:

  • Checked pod logs and identified frequent OOM kills.
  • Reviewed resource requests and limits in the pod spec.

  Root Cause: Resource limits were too low, leading to the container being killed when it exceeded available memory.

- Fix/Workaround:

• Increased the memory limits and requests for the affected pods.

• Re-deployed the updated pods and monitored for stability.

- Lessons Learned:

Proper resource requests and limits should be set to avoid OOM kills and pod restarts.

- How to Avoid:

• Regularly review resource requests and limits based on workload requirements.

• Use resource usage metrics to set more accurate resource limits.

Scenario #65: Cluster Performance Degradation Due to Excessive Logs

- Category: Cluster Management

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Cluster performance degraded because of excessive logs being generated by applications, leading to high disk usage.

- What Happened:Excessive log output from applications filled up the disk, slowing down the entire cluster.

- Diagnosis Steps:

• Monitored disk usage and found that logs were consuming most of the disk space.

• Identified the affected applications by reviewing the logging configuration.

Root Cause: Applications were logging excessively, and log rotation was not properly configured.

- Fix/Workaround:

• Configured log rotation for the affected applications.

• Reduced the verbosity of the logs in application settings.

- Lessons Learned:

Proper log management and rotation are essential to avoid filling up disk space and impacting cluster performance.

- How to Avoid:

• Configure log rotation and retention policies for all applications.

• Monitor disk usage and set up alerts for high usage.

Scenario #66: Insufficient Cluster Capacity Due to Unchecked CronJobs

- Category: Cluster Management

- Environment: K8s v1.21, GKE

Scenario

- Summary: The cluster experienced resource exhaustion because CronJobs were running in parallel without proper capacity checks.

- What Happened:Several CronJobs were triggered simultaneously, causing the cluster to run out of CPU and memory resources.

- Diagnosis Steps:

  • Checked CronJob schedules and found multiple jobs running at the same time.

  • Monitored resource usage and identified high CPU and memory consumption from the CronJobs.

  Root Cause: Lack of resource limits and concurrent job checks in CronJobs.

- Fix/Workaround:

  • Added resource requests and limits for CronJobs.

  • Configured CronJobs to stagger their execution times to avoid simultaneous execution.

- Lessons Learned:

  Always add resource limits and configure CronJobs to prevent them from running in parallel and exhausting cluster resources.

- How to Avoid:

  • Set appropriate resource requests and limits for CronJobs.

  • Use concurrencyPolicy to control parallel executions of CronJobs.

Scenario #67: Unsuccessful Pod Scaling Due to Affinity/Anti-Affinity Conflict

- Category: Cluster Management

- Environment: K8s v1.23, Azure AKS

  Scenario

- Summary: Pod scaling failed due to conflicting affinity/anti-affinity rules that prevented pods from being scheduled.

- What Happened:A deployment's pod scaling was unsuccessful due to the anti-affinity rules that conflicted with available nodes.

- Diagnosis Steps:

  • Checked pod scaling logs and identified unschedulable errors related to affinity rules.

  • Reviewed affinity/anti-affinity settings in the pod deployment configuration.

  Root Cause: The anti-affinity rule required pods to be scheduled on specific nodes, but there were not enough available nodes.

- Fix/Workaround:

  • Relaxed the anti-affinity rule to allow pods to be scheduled on any available node.

  • Increased the number of nodes to ensure sufficient capacity.

- Lessons Learned:

Affinity and anti-affinity rules should be configured carefully, especially in dynamic environments with changing node capacity.

- How to Avoid:

  • Test affinity and anti-affinity configurations thoroughly.

  • Use flexible affinity rules to allow for dynamic scaling and node availability.

Scenario #68: Cluster Inaccessibility Due to API Server Throttling

- Category: Cluster Management

- Environment: K8s v1.22, AWS EKS

  Scenario

- Summary: Cluster became inaccessible due to excessive API server throttling caused by too many concurrent requests.

- What Happened:The API server started throttling requests because the number of concurrent API calls exceeded the available limit.

- Diagnosis Steps:

  • Monitored API server metrics and identified a high rate of incoming requests.

  • Checked client application logs and observed excessive API calls.

  Root Cause: Clients were making too many API requests in a short period, exceeding the rate limits of the API server.

- Fix/Workaround:

• Throttled client requests to reduce API server load.

• Implemented exponential backoff for retries in client applications.

- Lessons Learned:

Avoid overwhelming the API server with excessive requests and implement rate-limiting mechanisms.

- How to Avoid:

• Implement API request throttling and retries in client applications.

• Use rate-limiting tools like kubectl to monitor API usage.

Scenario #69: Persistent Volume Expansion Failure

- Category: Cluster Management

- Environment: K8s v1.20, GKE

Scenario

- Summary: Expansion of a Persistent Volume (PV) failed due to improper storage class settings.

- What Happened:The request to expand a persistent volume failed because the storage class was not configured to support volume expansion.

- Diagnosis Steps:

  • Verified the PV and PVC configurations.

  • Checked the storage class settings and identified that volume expansion was not enabled.

  Root Cause: The storage class did not have the allowVolumeExpansion flag set to true.

- Fix/Workaround:

  • Updated the storage class to allow volume expansion.

  • Expanded the persistent volume and verified the PVC reflected the changes.

- Lessons Learned:

  Ensure that storage classes are configured to allow volume expansion when using dynamic provisioning.

- How to Avoid:

  • Check storage class configurations before creating PVs.

  • Enable allowVolumeExpansion for dynamic storage provisioning.

Scenario #70: Unauthorized Access to Cluster Resources Due to RBAC Misconfiguration

- Category: Cluster Management

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Unauthorized users gained access to sensitive resources due to misconfigured RBAC roles and bindings.

- What Happened:An RBAC misconfiguration allowed unauthorized users to access cluster resources, including secrets.

- Diagnosis Steps:

  • Checked RBAC policies and found overly permissive role bindings.

  • Analyzed user access logs and identified unauthorized access to sensitive resources.

Root Cause: Over-permissive RBAC role bindings granted excessive access to unauthorized users.

- Fix/Workaround:

  • Corrected RBAC policies to restrict access.

  • Audited user access and removed unauthorized permissions.

- Lessons Learned:

Proper RBAC configuration is crucial for securing cluster resources.

- How to Avoid:

  • Implement the principle of least privilege for RBAC roles.

  • Regularly audit RBAC policies and bindings.

Scenario #71: Inconsistent Pod State Due to Image Pull Failures

- Category: Cluster Management

- Environment: K8s v1.20, GKE

  Scenario

- Summary: Pods entered an inconsistent state because the container image failed to pull due to incorrect image tag.

- What Happened:Pods started with an image tag that did not exist in the container registry, causing the pods to enter a CrashLoopBackOff state.

- Diagnosis Steps:

  • Checked the pod events and found image pull errors with "Tag not found" messages.

  • Verified the image tag in the deployment configuration.

  Root Cause: The container image tag specified in the deployment was incorrect or missing from the container registry.

- Fix/Workaround:

  • Corrected the image tag in the deployment configuration to point to an existing image.

  • Redeployed the application.

- Lessons Learned:

Always verify image tags before deploying and ensure the image is available in the registry.

- How to Avoid:

  • Use CI/CD pipelines to automatically verify image availability before deployment.

  • Enable image pull retries for transient network issues.

Scenario #72: Pod Disruption Due to Insufficient Node Resources

- Category: Cluster Management

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: Pods experienced disruptions as nodes ran out of CPU and memory, causing evictions.

- What Happened:During a high workload period, nodes ran out of resources, causing the scheduler to evict pods and causing disruptions.

- Diagnosis Steps:

  • Monitored node resource usage and identified CPU and memory exhaustion.

  • Reviewed pod events and noticed pod evictions due to resource pressure.

Root Cause: Insufficient node resources for the workload being run, causing resource contention and pod evictions.

- Fix/Workaround:

  • Added more nodes to the cluster to meet resource requirements.

  • Adjusted pod resource requests/limits to be more aligned with node resources.

- Lessons Learned:

  Regularly monitor and scale nodes to ensure sufficient resources during peak workloads.

- How to Avoid:

  • Use cluster autoscaling to add nodes automatically when resource pressure increases.

  • Set appropriate resource requests and limits for pods.

Scenario #73: Service Discovery Issues Due to DNS Resolution Failures

- Category: Cluster Management

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Services could not discover each other due to DNS resolution failures, affecting internal communication.

- What Happened:Pods were unable to resolve internal service names due to DNS failures, leading to broken inter-service communication.

- Diagnosis Steps:

  • Checked DNS logs and found dnsmasq errors.

  • Investigated CoreDNS logs and found insufficient resources allocated to the DNS pods.

  Root Cause: CoreDNS pods were running out of resources (CPU/memory), causing DNS resolution failures.

- Fix/Workaround:

  • Increased resource limits for the CoreDNS pods.

  • Restarted CoreDNS pods to apply the new resource settings.

- Lessons Learned:

  Ensure that CoreDNS has enough resources to handle DNS requests efficiently.

- How to Avoid:

  • Monitor CoreDNS pod resource usage.

  • Allocate adequate resources based on cluster size and workload.

Scenario #74: Persistent Volume Provisioning Delays

- Category: Cluster Management

- Environment: K8s v1.22, GKE

Scenario

- Summary: Persistent volume provisioning was delayed due to an issue with the dynamic provisioner.

- What Happened:PVCs were stuck in the Pending state because the dynamic provisioner could not create the required PVs.

- Diagnosis Steps:

• Checked PVC status using kubectl get pvc and saw that they were stuck in Pending.

• Investigated storage class settings and found an issue with the provisioner configuration.

Root Cause: Misconfigured storage class settings were preventing the dynamic provisioner from provisioning volumes.

- Fix/Workaround:

• Corrected the storage class settings, ensuring the correct provisioner was specified.

• Recreated the PVCs, and provisioning completed successfully.

- Lessons Learned:

Validate storage class settings and provisioner configurations during cluster setup.

- How to Avoid:

• Test storage classes and volume provisioning in staging environments before production use.

• Monitor PV provisioning and automate alerts for failures.

Scenario #75: Deployment Rollback Failure Due to Missing Image

- Category: Cluster Management

- Environment: K8s v1.21, Azure AKS

Scenario

- Summary: A deployment rollback failed due to the rollback image version no longer being available in the container registry.

- What Happened:After an update, the deployment was rolled back to a previous image version that was no longer present in the container registry, causing the rollback to fail.

- Diagnosis Steps:

• Checked the deployment history and found that the previous image was no longer available.

• Examined the container registry and confirmed the image version had been deleted.

Root Cause: The image version intended for rollback was deleted from the registry before the rollback occurred.

- Fix/Workaround:

• Rebuilt the previous image version and pushed it to the registry.

• Triggered a successful rollback after the image was available.

- Lessons Learned:

Always retain previous image versions for safe rollbacks.

- How to Avoid:

• Implement retention policies for container images.

• Use CI/CD pipelines to tag and store images for future rollbacks.

Scenario #76: Kubernetes Master Node Unresponsive After High Load

- Category: Cluster Management

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: The Kubernetes master node became unresponsive under high load due to excessive API server calls and high memory usage.

- What Happened:The Kubernetes master node was overwhelmed by API calls and high memory consumption, leading to a failure to respond to management requests.

- Diagnosis Steps:

• Checked the control plane resource usage and found high memory and CPU consumption on the master node.

• Analyzed API server logs and found a spike in incoming requests.

Root Cause: Excessive incoming requests caused API server memory to spike, rendering the master node unresponsive.

- Fix/Workaround:

• Implemented API rate limiting to control excessive calls.

• Increased the memory allocated to the master node.

- Lessons Learned:

Ensure that the control plane is protected against overloads and is properly scaled.

- How to Avoid:

• Use API rate limiting and load balancing techniques for the master node.

• Consider separating the control plane and worker nodes for better scalability.

Scenario #77: Failed Pod Restart Due to Inadequate Node Affinity

- Category: Cluster Management

- Environment: K8s v1.24, GKE

Scenario

- Summary: Pods failed to restart on available nodes due to overly strict node affinity rules.

- What Happened:A pod failed to restart after a node failure because the node affinity rules were too strict, preventing the pod from being scheduled on any available nodes.

- Diagnosis Steps:

  • Checked pod logs and observed affinity errors in scheduling.

  • Analyzed the affinity settings in the pod spec and found restrictive affinity rules.

  Root Cause: Strict node affinity rules prevented the pod from being scheduled on available nodes.

- Fix/Workaround:

  • Relaxed the node affinity rules in the pod spec.

  • Redeployed the pod, and it successfully restarted on an available node.

- Lessons Learned:

  Carefully configure node affinity rules to allow flexibility during pod rescheduling.

- How to Avoid:

  • Use less restrictive affinity rules for better pod rescheduling flexibility.

  • Test affinity rules during node maintenance and scaling operations.

Scenario #78: ReplicaSet Scaling Issues Due to Resource Limits

- Category: Cluster Management

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: The ReplicaSet failed to scale due to insufficient resources on the nodes.

- What Happened:When attempting to scale a ReplicaSet, new pods failed to schedule due to a lack of available resources on the nodes.

- Diagnosis Steps:

• Checked the resource usage on the nodes and found they were running at full capacity.

• Analyzed ReplicaSet scaling events and observed failures to schedule new pods.

Root Cause: Insufficient node resources to accommodate new pods due to high resource consumption by existing workloads.

- Fix/Workaround:

• Added more nodes to the cluster to handle the increased workload.

• Adjusted resource requests and limits to ensure efficient resource allocation.

- Lessons Learned:

Regularly monitor cluster resource usage and scale proactively based on demand.

- How to Avoid:

• Enable cluster autoscaling to handle scaling issues automatically.

• Set proper resource requests and limits for pods to avoid resource exhaustion.

Scenario #79: Missing Namespace After Cluster Upgrade

- Category: Cluster Management

- Environment: K8s v1.21, GKE

Scenario

- Summary: A namespace was missing after performing a cluster upgrade.

- What Happened:After upgrading the cluster, a namespace that was present before the upgrade was no longer available.

- Diagnosis Steps:

• Checked the cluster upgrade logs and identified that a namespace deletion had occurred during the upgrade process.

• Verified with backup and confirmed the namespace was inadvertently deleted during the upgrade.

Root Cause: An issue during the cluster upgrade process led to the unintentional deletion of a namespace.

- Fix/Workaround:

• Restored the missing namespace from backups.

• Investigated and fixed the upgrade process to prevent future occurrences.

- Lessons Learned:

Always backup critical resources before performing upgrades and test the upgrade process thoroughly.

- How to Avoid:

• Backup namespaces and other critical resources before upgrading.

• Review upgrade logs carefully to identify any unexpected deletions or changes.

Scenario #80: Inefficient Resource Usage Due to Misconfigured Horizontal Pod Autoscaler

- Category: Cluster Management

- Environment: K8s v1.23, Azure AKS

Scenario

- Summary: The Horizontal Pod Autoscaler (HPA) was inefficiently scaling due to misconfigured metrics.

- What Happened:HPA did not scale pods appropriately, either under-scaling or over-scaling, due to incorrect metric definitions.

- Diagnosis Steps:

• Checked HPA configuration and identified incorrect CPU utilization metrics.

• Monitored metrics-server logs and found that the metrics were inconsistent.

Root Cause: HPA was configured to scale based on inaccurate or inappropriate metrics, leading to inefficient scaling behavior.

- Fix/Workaround:

• Reconfigured the HPA to scale based on correct metrics (e.g., memory, custom metrics).

• Verified that the metrics-server was reporting accurate data.

- Lessons Learned:

Always ensure that the right metrics are used for scaling to avoid inefficient scaling behavior.

- How to Avoid:

• Regularly review HPA configurations and metrics definitions.

• Test scaling behavior under different load conditions.

Scenario #81: Pod Disruption Due to Unavailable Image Registry

- Category: Cluster Management

- Environment: K8s v1.21, GKE

Scenario

- Summary: Pods could not start because the image registry was temporarily unavailable, causing image pull failures.

- What Happened:Pods failed to pull images because the registry was down for maintenance, leading to deployment failures.

- Diagnosis Steps:

  • Checked the pod status using kubectl describe pod and identified image pull errors.

  • Investigated the registry status and found scheduled downtime for maintenance.

  Root Cause: The container registry was temporarily unavailable due to maintenance, and the pods could not access the required images.

- Fix/Workaround:

  • Manually downloaded the images from a secondary registry.

  • Temporarily used a local image registry until the primary registry was back online.

- Lessons Learned:

  Ensure that alternate image registries are available in case of downtime.

- How to Avoid:

  • Implement multiple image registries for high availability.

  • Use image pull policies that allow fallback to local caches.

Scenario #82: Pod Fails to Start Due to Insufficient Resource Requests

- Category: Cluster Management

- Environment: K8s v1.20, AWS EKS

Scenario

- Summary: Pods failed to start because their resource requests were too low, preventing the scheduler from assigning them to nodes.

- What Happened:The pods had very low resource requests, causing the scheduler to fail to assign them to available nodes.

- Diagnosis Steps:

  • Checked pod status and found them stuck in Pending.
  • Analyzed the resource requests and found that they were too low to meet the node's capacity requirements.

Root Cause: The resource requests were set too low, preventing proper pod scheduling.

- Fix/Workaround:

  • Increased the resource requests in the pod spec.
  • Reapplied the configuration, and the pods were scheduled successfully.

- Lessons Learned:

Always ensure that resource requests are appropriately set for your workloads.

- How to Avoid:

• Use resource limits and requests based on accurate usage data from monitoring tools.

• Set resource requests in line with expected workload sizes.

Scenario #83: Horizontal Pod Autoscaler Under-Scaling During Peak Load

- Category: Cluster Management

- Environment: K8s v1.22, GKE

Scenario

- Summary: HPA failed to scale the pods appropriately during a sudden spike in load.

- What Happened:The HPA did not scale the pods properly during a traffic surge due to incorrect metric thresholds.

- Diagnosis Steps:

• Checked HPA settings and identified that the CPU utilization threshold was too high.

• Verified the metric server was reporting correct metrics.

Root Cause: Incorrect scaling thresholds set in the HPA configuration.

- Fix/Workaround:

• Adjusted HPA thresholds to scale more aggressively under higher loads.

• Increased the replica count to handle the peak load.

- Lessons Learned:

HPA thresholds should be fine-tuned based on expected load patterns.

- How to Avoid:

• Regularly review and adjust HPA configurations to reflect actual workload behavior.

• Use custom metrics for better scaling control.

Scenario #84: Pod Eviction Due to Node Disk Pressure

- Category: Cluster Management

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Pods were evicted due to disk pressure on the node, causing service interruptions.

- What Happened:A node ran out of disk space due to logs and other data consuming the disk, resulting in pod evictions.

- Diagnosis Steps:

• Checked node resource usage and found disk space was exhausted.

• Reviewed pod eviction events and found they were due to disk pressure.

Root Cause: The node disk was full, causing the kubelet to evict pods to free up resources.

- Fix/Workaround:

  • Increased disk capacity on the affected node.
  • Cleared unnecessary logs and old data from the disk.

- Lessons Learned:

  Ensure adequate disk space is available, especially for logging and temporary data.

- How to Avoid:

  • Monitor disk usage closely and set up alerts for disk pressure.
  • Implement log rotation and clean-up policies to avoid disk exhaustion.

Scenario #85: Failed Node Drain Due to In-Use Pods

- Category: Cluster Management

- Environment: K8s v1.22, Azure AKS

  Scenario

- Summary: A node failed to drain due to pods that were in use, preventing the drain operation from completing.

- What Happened:When attempting to drain a node, the operation failed because some pods were still in use or had pending termination grace periods.

- Diagnosis Steps:

  • Ran kubectl describe node and checked pod evictions.

  • Identified pods that were in the middle of long-running processes or had insufficient termination grace periods.

  Root Cause: Pods with long-running tasks or improper termination grace periods caused the drain to hang.

- Fix/Workaround:

  • Increased termination grace periods for the affected pods.

  • Forced the node drain operation after ensuring that the pods could safely terminate.

- Lessons Learned:

  Ensure that pods with long-running tasks have adequate termination grace periods.

- How to Avoid:

  • Configure appropriate termination grace periods for all pods.

  • Monitor node draining and ensure pods can gracefully shut down.

Scenario #86: Cluster Autoscaler Not Scaling Up

- Category: Cluster Management

- Environment: K8s v1.20, GKE

  Scenario

- Summary: The cluster autoscaler failed to scale up the node pool despite high resource demand.

- What Happened:The cluster autoscaler did not add nodes when resource utilization reached critical levels.

- Diagnosis Steps:

  • Checked the autoscaler logs and found that scaling events were not triggered.

  • Reviewed the node pool configuration and autoscaler settings.

  Root Cause: The autoscaler was not configured with sufficient thresholds or permissions to scale up the node pool.

- Fix/Workaround:

  • Adjusted the scaling thresholds in the autoscaler configuration.

  • Verified the correct IAM permissions for the autoscaler to scale the node pool.

- Lessons Learned:

  Ensure the cluster autoscaler is correctly configured and has the right permissions.

- How to Avoid:

  • Regularly review cluster autoscaler configuration and permissions.

  • Monitor scaling behavior to ensure it functions as expected during high load.

Scenario #87: Pod Network Connectivity Issues After Node Reboot

- Category: Cluster Management

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Pods lost network connectivity after a node reboot, causing communication failures between services.

- What Happened:After a node was rebooted, the networking components failed to re-establish proper connectivity for the pods.

- Diagnosis Steps:

• Checked pod logs and found connection timeouts between services.

• Investigated the node and found networking components (e.g., CNI plugin) were not properly re-initialized after the reboot.

Root Cause: The CNI plugin did not properly re-initialize after the node reboot, causing networking failures.

- Fix/Workaround:

• Manually restarted the CNI plugin on the affected node.

• Ensured that the CNI plugin was configured to restart properly after a node reboot.

- Lessons Learned:

Ensure that critical components like CNI plugins are resilient to node reboots.

- How to Avoid:

  • Configure the CNI plugin to restart automatically after node reboots.

  • Monitor networking components to ensure they are healthy after reboots.

Scenario #88: Insufficient Permissions Leading to Unauthorized Access Errors

- Category: Cluster Management

- Environment: K8s v1.22, GKE

Scenario

- Summary: Unauthorized access errors occurred due to missing permissions in RBAC configurations.

- What Happened:Pods failed to access necessary resources due to misconfigured RBAC policies, resulting in permission-denied errors.

- Diagnosis Steps:

  • Reviewed the RBAC policy logs and identified missing permissions for service accounts.

  • Checked the roles and role bindings associated with the pods.

Root Cause: RBAC policies did not grant the required permissions to the service accounts.

- Fix/Workaround:

• Updated the RBAC roles and bindings to include the necessary permissions for the pods.

• Applied the updated RBAC configurations and confirmed access.

- Lessons Learned:

RBAC configurations should be thoroughly tested to ensure correct permissions.

- How to Avoid:

• Implement a least-privilege access model and audit RBAC policies regularly.

• Use automated tools to test and verify RBAC configurations.

Scenario #89: Failed Pod Upgrade Due to Incompatible API Versions

- Category: Cluster Management

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: A pod upgrade failed because it was using deprecated APIs not supported in the new version.

- What Happened:When upgrading to a new Kubernetes version, a pod upgrade failed due to deprecated APIs in use.

- Diagnosis Steps:

  • Checked the pod spec and identified deprecated API versions in use.

  • Verified the Kubernetes changelog for API deprecations in the new version.

  Root Cause: The pod was using APIs that were deprecated in the new Kubernetes version, causing the upgrade to fail.

- Fix/Workaround:

  • Updated the pod spec to use supported API versions.

  • Reapplied the deployment with the updated APIs.

- Lessons Learned:

  Regularly review Kubernetes changelogs for deprecated API versions.

- How to Avoid:

  • Implement a process to upgrade and test all components for compatibility before applying changes.

  • Use tools like kubectl deprecations to identify deprecated APIs.

Scenario #90: High CPU Utilization Due to Inefficient Application Code

- Category: Cluster Management

- Environment: K8s v1.21, Azure AKS

Scenario

- Summary: A container's high CPU usage was caused by inefficient application code, leading to resource exhaustion.

- What Happened:An application was running inefficient code that caused excessive CPU consumption, impacting the entire node's performance.

- Diagnosis Steps:

• Monitored the pod's resource usage and found high CPU utilization.

• Analyzed application logs and identified inefficient loops in the code.

Root Cause: The application code had an inefficient algorithm that led to high CPU consumption.

- Fix/Workaround:

• Optimized the application code to reduce CPU consumption.

• Redeployed the application with the optimized code.

- Lessons Learned:

Application code optimization is essential for ensuring efficient resource usage.

- How to Avoid:

• Regularly profile application code for performance bottlenecks.

• Set CPU limits and requests to prevent resource exhaustion.

Scenario #91: Resource Starvation Due to Over-provisioned Pods

- Category: Cluster Management

- Environment: K8s v1.20, AWS EKS

Scenario

- Summary: Resource starvation occurred on nodes because pods were over-provisioned, consuming more resources than expected.

- What Happened:Pods were allocated more resources than necessary, causing resource contention on the nodes.

- Diagnosis Steps:

  • Analyzed node and pod resource utilization.

  • Found that the CPU and memory resources for several pods were unnecessarily high, leading to resource starvation for other pods.

Root Cause: Incorrect resource requests and limits set for the pods, causing resource over-allocation.

- Fix/Workaround:

  • Reduced resource requests and limits based on actual usage metrics.

  • Re-deployed the pods with optimized resource configurations.

- Lessons Learned:

Accurate resource requests and limits should be based on actual usage data.

- How to Avoid:

• Regularly monitor resource utilization and adjust requests/limits accordingly.

• Use vertical pod autoscalers for better resource distribution.

Scenario #92: Unscheduled Pods Due to Insufficient Affinity Constraints

- Category: Cluster Management

- Environment: K8s v1.21, GKE

Scenario

- Summary: Pods were not scheduled due to overly strict affinity rules that limited the nodes available for deployment.

- What Happened:The affinity rules were too restrictive, preventing pods from being scheduled on available nodes.

- Diagnosis Steps:

• Reviewed pod deployment spec and found strict affinity constraints.

• Verified available nodes and found that no nodes met the pod's affinity requirements.

Root Cause: Overly restrictive affinity settings that limited pod scheduling.

- Fix/Workaround:

  • Adjusted the affinity rules to be less restrictive.

  • Applied changes and verified the pods were scheduled correctly.

- Lessons Learned:

  Affinity constraints should balance optimal placement with available resources.

- How to Avoid:

  • Regularly review and adjust affinity/anti-affinity rules based on cluster capacity.

  • Test deployment configurations in staging before applying to production.

Scenario #93: Pod Readiness Probe Failure Due to Slow Initialization

- Category: Cluster Management

- Environment: K8s v1.22, Azure AKS

  Scenario

- Summary: Pods failed their readiness probes during initialization, causing traffic to be routed to unhealthy instances.

- What Happened:The pods had a slow initialization time, but the readiness probe timeout was set too low, causing premature failure.

- Diagnosis Steps:

• Checked pod events and logs, discovering that readiness probes were failing due to long startup times.

• Increased the timeout period for the readiness probe and observed that the pods began to pass the probe after startup.

Root Cause: Readiness probe timeout was set too low for the pod's initialization process.

- Fix/Workaround:

• Increased the readiness probe timeout and delay parameters.

• Re-applied the deployment, and the pods started passing readiness checks.

- Lessons Learned:

The readiness probe timeout should be configured according to the actual initialization time of the pod.

- How to Avoid:

• Monitor pod initialization times and adjust readiness probe configurations accordingly.

• Use a gradual rollout for new deployments to avoid sudden failures.

Scenario #94: Incorrect Ingress Path Handling Leading to 404 Errors

- Category: Cluster Management

- Environment: K8s v1.19, GKE

  Scenario

- Summary: Incorrect path configuration in the ingress resource resulted in 404 errors for certain API routes.

- What Happened:Ingress was misconfigured with incorrect path mappings, causing requests to certain API routes to return 404 errors.

- Diagnosis Steps:

  • Checked ingress configuration using kubectl describe ingress and found mismatched path rules.

  • Verified the service endpoints and found that the routes were not properly configured in the ingress spec.

  Root Cause: Incorrect path definitions in the ingress resource, causing requests to be routed incorrectly.

- Fix/Workaround:

  • Fixed the path configuration in the ingress resource.

  • Re-applied the ingress configuration, and traffic was correctly routed.

- Lessons Learned:

  Verify that ingress path definitions match the application routing.

- How to Avoid:

  • Test ingress paths thoroughly before applying to production environments.

• Use versioned APIs to ensure backward compatibility for routing paths.

Scenario #95: Node Pool Scaling Failure Due to Insufficient Quotas

- Category: Cluster Management

- Environment: K8s v1.20, AWS EKS

Scenario

- Summary: Node pool scaling failed because the account exceeded resource quotas in AWS.

- What Happened:When attempting to scale up a node pool, the scaling operation failed due to hitting AWS resource quotas.

- Diagnosis Steps:

• Reviewed the EKS and AWS console to identify quota limits.

• Found that the account had exceeded the EC2 instance limit for the region.

Root Cause: Insufficient resource quotas in the AWS account.

- Fix/Workaround:

• Requested a quota increase from AWS support.

• Once the request was approved, scaled the node pool successfully.

- Lessons Learned:

Monitor cloud resource quotas to ensure scaling operations are not blocked.

- How to Avoid:

• Keep track of resource quotas and request increases in advance.

• Automate quota monitoring and alerting to avoid surprises during scaling.

Scenario #96: Pod Crash Loop Due to Missing ConfigMap

- Category: Cluster Management

- Environment: K8s v1.21, Azure AKS

Scenario

- Summary: Pods entered a crash loop because a required ConfigMap was not present in the namespace.

- What Happened:A pod configuration required a ConfigMap that was deleted by accident, causing the pod to crash due to missing configuration data.

- Diagnosis Steps:

• Checked pod logs and found errors indicating missing environment variables or configuration files.

• Investigated the ConfigMap and found it had been accidentally deleted.

Root Cause: Missing ConfigMap due to accidental deletion.

- Fix/Workaround:

• Recreated the ConfigMap in the namespace.

• Re-deployed the pods, and they started successfully.

- Lessons Learned:

Protect critical resources like ConfigMaps to prevent accidental deletion.

- How to Avoid:

• Use namespaces and resource quotas to limit accidental deletion of shared resources.

• Implement stricter RBAC policies for sensitive resources.

Scenario #97: Kubernetes API Server Slowness Due to Excessive Logging

- Category: Cluster Management

- Environment: K8s v1.22, GKE

Scenario

- Summary: The Kubernetes API server became slow due to excessive log generation from the kubelet and other components.

- What Happened:Excessive logging from the kubelet and other components overwhelmed the API server, causing it to become slow and unresponsive.

- Diagnosis Steps:

  • Monitored API server performance using kubectl top pod and noticed resource spikes.

  • Analyzed log files and found an unusually high number of log entries from the kubelet.

  Root Cause: Excessive logging was causing resource exhaustion on the API server.

- Fix/Workaround:

  • Reduced the verbosity of logs from the kubelet and other components.

  • Configured log rotation to prevent logs from consuming too much disk space.

- Lessons Learned:

  Excessive logging can cause performance degradation if not properly managed.

- How to Avoid:

  • Set appropriate logging levels for components based on usage.

  • Implement log rotation and retention policies to avoid overwhelming storage.

Scenario #98: Pod Scheduling Failure Due to Taints and Tolerations Misconfiguration

- Category: Cluster Management

- Environment: K8s v1.19, AWS EKS

  Scenario

- Summary: Pods failed to schedule because the taints and tolerations were misconfigured, preventing the scheduler from placing them on nodes.

- What Happened:The nodes had taints that were not matched by the pod's tolerations, causing the pods to remain unscheduled.

- Diagnosis Steps:

  • Used kubectl describe pod to investigate scheduling issues.
  • Found that the taints on the nodes did not match the tolerations set on the pods.

  Root Cause: Misconfiguration of taints and tolerations in the node and pod specs.

- Fix/Workaround:

  • Corrected the tolerations in the pod specs to match the taints on the nodes.
  • Re-applied the pods and verified that they were scheduled correctly.

- Lessons Learned:

  Always ensure taints and tolerations are correctly configured in a multi-tenant environment.

- How to Avoid:

  • Test taints and tolerations in a non-production environment.
  • Regularly audit and verify toleration settings to ensure proper pod placement.

Scenario #99: Unresponsive Dashboard Due to High Resource Usage

- Category: Cluster Management

- Environment: K8s v1.20, Azure AKS

  Scenario

- Summary: The Kubernetes dashboard became unresponsive due to high resource usage caused by a large number of requests.

- What Happened:The Kubernetes dashboard was overwhelmed by too many requests, consuming excessive CPU and memory resources.

- Diagnosis Steps:

  • Checked resource usage of the dashboard pod using kubectl top pod.

  • Found that the pod was using more resources than expected due to a large number of incoming requests.

  Root Cause: The dashboard was not scaled to handle the volume of requests.

- Fix/Workaround:

  • Scaled the dashboard deployment to multiple replicas to handle the load.

  • Adjusted resource requests and limits for the dashboard pod.

- Lessons Learned:

  Ensure that the Kubernetes dashboard is properly scaled to handle expected traffic.

- How to Avoid:

• Implement horizontal scaling for the dashboard and other critical services.

• Monitor the usage of the Kubernetes dashboard and scale as needed.

Scenario #100: Resource Limits Causing Container Crashes

- Category: Cluster Management

- Environment: K8s v1.21, GKE

Scenario

- Summary: Containers kept crashing due to hitting resource limits set in their configurations.

- What Happened:Containers were being killed because they exceeded their resource limits for memory and CPU.

- Diagnosis Steps:

• Used kubectl describe pod to find the resource limits and found that the limits were too low for the workload.

• Analyzed container logs and found frequent OOMKilled events.

Root Cause: The resource limits set for the container were too low, causing the container to be terminated when it exceeded the limit.

- Fix/Workaround:

• Increased the resource limits for the affected containers.

• Re-applied the pod configurations and monitored for stability.

- Lessons Learned:

Resource limits should be set based on actual workload requirements.

- How to Avoid:

• Use monitoring tools to track resource usage and adjust limits as needed.

• Set up alerts for resource threshold breaches to avoid crashes.

NETWORKING

Scenario #101: Pod Communication Failure Due to Network Policy Misconfiguration

- Category: Networking

- Environment: K8s v1.22, GKE

Scenario

- Summary: Pods failed to communicate due to a misconfigured NetworkPolicy that blocked ingress traffic.

- What Happened:A newly applied NetworkPolicy was too restrictive, preventing communication between certain pods within the same namespace.

- Diagnosis Steps:

• Used kubectl get networkpolicies to inspect the NetworkPolicy.

• Identified that the ingress rules were overly restrictive and did not allow traffic between pods that needed to communicate.

Root Cause: The NetworkPolicy did not account for the required communication between pods.

- Fix/Workaround:

• Updated the NetworkPolicy to allow the necessary ingress traffic between the affected pods.

• Re-applied the NetworkPolicy and tested communication.

- Lessons Learned:

Network policies need to be tested thoroughly, especially in multi-tenant or complex networking environments.

- How to Avoid:

• Use staging environments to test NetworkPolicy changes.

• Apply policies incrementally and monitor network traffic.

Scenario #102: DNS Resolution Failure Due to CoreDNS Pod Crash

- Category: Networking

- Environment: K8s v1.21, Azure AKS

Scenario

- Summary: DNS resolution failed across the cluster after CoreDNS pods crashed unexpectedly.

- What Happened:CoreDNS pods were crashed due to resource exhaustion, leading to DNS resolution failure for all services.

- Diagnosis Steps:

• Used kubectl get pods -n kube-system to check the status of CoreDNS pods.

• Found that CoreDNS pods were in a crash loop due to memory resource limits being set too low.

Root Cause: CoreDNS resource limits were too restrictive, causing it to run out of memory.

- Fix/Workaround:

• Increased memory limits for CoreDNS pods.

• Restarted the CoreDNS pods and verified DNS resolution functionality.

- Lessons Learned:

Ensure CoreDNS has sufficient resources to handle DNS queries for large clusters.

- How to Avoid:

• Regularly monitor CoreDNS metrics for memory and CPU usage.

• Adjust resource limits based on cluster size and traffic patterns.

Scenario #103: Network Latency Due to Misconfigured Service Type

- Category: Networking

- Environment: K8s v1.18, AWS EKS

Scenario

- Summary: High network latency occurred because a service was incorrectly configured as a NodePortinstead of a LoadBalancer.

- What Happened:Services behind a NodePort experienced high latency due to traffic being routed through each node instead of through an optimized load balancer.

- Diagnosis Steps:

• Checked the service configuration and identified that the service type was set to NodePort.

• Verified that traffic was hitting every node, causing uneven load distribution and high latency.

Root Cause: Incorrect service type that did not provide efficient load balancing.

- Fix/Workaround:

• Changed the service type to LoadBalancer, which properly routed traffic through a managed load balancer.

• Traffic was distributed evenly, and latency was reduced.

- Lessons Learned:

Choose the correct service type based on traffic patterns and resource requirements.

- How to Avoid:

  • Review service types based on the expected traffic and scalability.

  • Use a LoadBalancer for production environments requiring high availability.

Scenario #104: Inconsistent Pod-to-Pod Communication Due to MTU Mismatch

- Category: Networking

- Environment: K8s v1.20, GKE

Scenario

- Summary: Pod-to-pod communication became inconsistent due to a mismatch in Maximum Transmission Unit (MTU) settings across nodes.

- What Happened:Network packets were being fragmented or dropped due to inconsistent MTU settings between nodes.

- Diagnosis Steps:

  • Verified MTU settings on each node using ifconfig and noticed discrepancies between nodes.

  • Used ping with varying packet sizes to identify where fragmentation or packet loss occurred.

Root Cause: MTU mismatch between nodes and network interfaces.

- Fix/Workaround:

• Aligned MTU settings across all nodes in the cluster.

• Rebooted the nodes to apply the new MTU configuration.

- Lessons Learned:

Consistent MTU settings are crucial for reliable network communication.

- How to Avoid:

• Ensure that MTU settings are consistent across all network interfaces in the cluster.

• Test network connectivity regularly to ensure that no fragmentation occurs.

Scenario #105: Service Discovery Failure Due to DNS Pod Resource Limits

- Category: Networking

- Environment: K8s v1.19, Azure AKS

Scenario

- Summary: Service discovery failed across the cluster due to DNS pod resource limits being exceeded.

- What Happened:The DNS service was unable to resolve names due to resource limits being hit on the CoreDNS pods, causing failures in service discovery.

- Diagnosis Steps:

  • Checked CoreDNS pod resource usage and logs, revealing that the memory limit was being exceeded.

  • Found that DNS requests were timing out, and pods were unable to discover services.

  Root Cause: CoreDNS pods hit resource limits, leading to failures in service resolution.

- Fix/Workaround:

  • Increased memory and CPU limits for CoreDNS pods.

  • Restarted CoreDNS pods and verified that DNS resolution was restored.

- Lessons Learned:

  Service discovery requires sufficient resources to avoid failure.

- How to Avoid:

  • Regularly monitor CoreDNS metrics and adjust resource limits accordingly.

  • Scale CoreDNS replicas based on cluster size and traffic.

Scenario #106: Pod IP Collision Due to Insufficient IP Range

- Category: Networking

- Environment: K8s v1.21, GKE

Scenario

- Summary: Pod IP collisions occurred due to insufficient IP range allocation for the cluster.

- What Happened:Pods started having overlapping IPs, causing communication failures between pods.

- Diagnosis Steps:

  • Analyzed pod IPs and discovered that there were overlaps due to an insufficient IP range in the CNI plugin.

  • Identified that the IP range allocated during cluster creation was too small for the number of pods.

Root Cause: Incorrect IP range allocation when the cluster was initially created.

- Fix/Workaround:

  • Increased the pod network CIDR and restarted the cluster.

  • Re-deployed the affected pods to new IPs without collisions.

- Lessons Learned:

Plan IP ranges appropriately during cluster creation to accommodate scaling.

- How to Avoid:

  • Ensure that the IP range for pods is large enough to accommodate future scaling needs.

  • Monitor IP allocation and usage metrics for early detection of issues.

Scenario #107: Network Bottleneck Due to Single Node in NodePool

- Category: Networking

- Environment: K8s v1.23, AWS EKS

Scenario

- Summary: A network bottleneck occurred due to excessive traffic being handled by a single node in the node pool.

- What Happened:One node in the node pool was handling all the traffic for multiple pods, leading to CPU and network saturation.

- Diagnosis Steps:

• Checked node utilization with kubectl top node and identified a single node with high CPU and network load.

• Verified the load distribution across the node pool and found uneven traffic handling.

Root Cause: The cluster autoscaler did not scale the node pool correctly due to resource limits on the instance type.

- Fix/Workaround:

• Increased the size of the node pool and added more nodes with higher resource capacity.

• Rebalanced the pods across nodes and monitored for stability.

- Lessons Learned:

  Autoscaler configuration and node resource distribution are critical for handling high traffic.

- How to Avoid:

  • Ensure that the cluster autoscaler is correctly configured to balance resource load across all nodes.

  • Monitor traffic patterns and node utilization regularly.

Scenario #108: Network Partitioning Due to CNI Plugin Failure

- Category: Networking

- Environment: K8s v1.18, GKE

  Scenario

- Summary: A network partition occurred when the CNI plugin failed, preventing pods from communicating with each other.

- What Happened:The CNI plugin failed to configure networking correctly, causing network partitions within the cluster.

- Diagnosis Steps:

• Checked CNI plugin logs and found that the plugin was failing to initialize network interfaces for new pods.

• Verified pod network connectivity and found that they could not reach services in other namespaces.

Root Cause: Misconfiguration or failure of the CNI plugin, causing networking issues.

- Fix/Workaround:

• Reinstalled the CNI plugin and applied the correct network configuration.

• Re-deployed the affected pods after ensuring the network configuration was correct.

- Lessons Learned:

Ensure that the CNI plugin is properly configured and functioning.

- How to Avoid:

• Regularly test the CNI plugin and monitor logs for failures.

• Use redundant networking setups to avoid single points of failure.

Scenario #109: Misconfigured Ingress Resource Causing SSL Errors

- Category: Networking

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: SSL certificate errors occurred due to a misconfigured Ingress resource.

- What Happened:The Ingress resource had incorrect SSL certificate annotations, causing SSL handshake failures for external traffic.

- Diagnosis Steps:

  • Inspected Ingress resource configuration and identified the wrong certificate annotations.

  • Verified SSL errors in the logs, confirming SSL handshake issues.

  Root Cause: Incorrect SSL certificate annotations in the Ingress resource.

- Fix/Workaround:

  • Corrected the SSL certificate annotations in the Ingress configuration.

  • Re-applied the Ingress resource and verified successful SSL handshakes.

- Lessons Learned:

  Double-check SSL-related annotations and configurations for ingress resources.

- How to Avoid:

  • Use automated certificate management tools like cert-manager for better SSL certificate handling.

  • Test SSL connections before deploying ingress resources in production.

Scenario #110: Cluster Autoscaler Fails to Scale Nodes Due to Incorrect IAM Role Permissions

- Category: Cluster Management

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: The cluster autoscaler failed to scale the number of nodes in response to resource shortages due to missing IAM role permissions for managing EC2 instances.

- What Happened:The cluster autoscaler tried to add nodes to the cluster, but due to insufficient IAM permissions, it was unable to interact with EC2 to provision new instances. This led to insufficient resources, affecting pod scheduling.

- Diagnosis Steps:

• Checked kubectl describe pod and noted that pods were in pending state due to resource shortages.

• Analyzed the IAM roles and found that the permissions required by the cluster autoscaler to manage EC2 instances were missing.

Root Cause: Missing IAM role permissions for the cluster autoscaler prevented node scaling.

- Fix/Workaround:

• Updated the IAM role associated with the cluster autoscaler to include the necessary permissions for EC2 instance provisioning.

• Restarted the autoscaler and confirmed that new nodes were added successfully.

- Lessons Learned:

Ensure that the cluster autoscaler has the required permissions to scale nodes in cloud environments.

- How to Avoid:

• Regularly review IAM permissions and role configurations for essential services like the cluster autoscaler.

• Automate IAM permission audits to catch configuration issues early.

Scenario #111: DNS Resolution Failure Due to Incorrect Pod IP Allocation

- Category: Networking

- Environment: K8s v1.21, GKE

Scenario

- Summary: DNS resolution failed due to incorrect IP allocation in the cluster's CNI plugin.

- What Happened:Pods were allocated IPs outside the expected range, causing DNS queries to fail since the DNS service was not able to route correctly.

- Diagnosis Steps:

• Reviewed the IP range configuration for the CNI plugin and verified that IPs allocated to pods were outside the defined CIDR block.

• Observed that pods with incorrect IP addresses couldn't register with CoreDNS.

Root Cause: Misconfiguration of the CNI plugin's IP allocation settings.

- Fix/Workaround:

  • Reconfigured the CNI plugin to correctly allocate IPs within the defined range.
  • Re-deployed affected pods with new IPs that were correctly assigned.

- Lessons Learned:
  Always verify IP range configuration when setting up or scaling CNI plugins.

- How to Avoid:

  • Check IP allocation settings regularly and use monitoring tools to track IP usage.
  • Ensure CNI plugin configurations align with network architecture requirements.

Scenario #112: Failed Pod-to-Service Communication Due to Port Binding Conflict

- Category: Networking

- Environment: K8s v1.18, AWS EKS
  Scenario

- Summary: Pods couldn't communicate with services because of a port binding conflict.

- What Happened:A service was configured with a port that was already in use by another pod, causing connectivity issues.

- Diagnosis Steps:

• Inspected service and pod configurations using kubectl describe to identify the port conflict.

• Found that the service port conflicted with the port used by a previously deployed pod.

Root Cause: Port binding conflict caused the service to be unreachable from the pod.

- Fix/Workaround:

• Changed the port for the service to a free port and re-applied the service configuration.

• Verified that pod communication was restored.

- Lessons Learned:

Properly manage port allocations and avoid conflicts.

- How to Avoid:

• Use port management strategies and avoid hardcoding ports in services and pods.

• Automate port management and checking within deployment pipelines.

Scenario #113: Pod Eviction Due to Network Resource Constraints

- Category: Networking

- Environment: K8s v1.19, GKE

Scenario

- Summary: A pod was evicted due to network resource constraints, specifically limited bandwidth.

- What Happened:The pod was evicted by the kubelet due to network resource limits being exceeded, leading to a failure in service availability.

- Diagnosis Steps:

  • Used kubectl describe pod to investigate the eviction event and noted network-related resource constraints in the pod eviction message.

  • Checked node network resource limits and found bandwidth throttling was causing evictions.

Root Cause: Insufficient network bandwidth allocation for the pod.

- Fix/Workaround:

  • Increased network bandwidth limits on the affected node pool.

  • Re-scheduled the pod on a node with higher bandwidth availability.

- Lessons Learned:

Network bandwidth limits can impact pod availability and performance.

- How to Avoid:

  • Monitor and adjust network resource allocations regularly.

  • Use appropriate pod resource requests and limits to prevent evictions.

Scenario #114: Intermittent Network Disconnects Due to MTU Mismatch Between Nodes

- Category: Networking

- Environment: K8s v1.20, Azure AKS

  Scenario

- Summary: Intermittent network disconnects occurred due to MTU mismatches between different nodes in the cluster.

- What Happened:Network packets were being dropped or fragmented between nodes with different MTU settings, causing network instability.

- Diagnosis Steps:

  • Used ping with large payloads to identify packet loss.

  • Discovered that the MTU was mismatched between the nodes and the network interface.

  Root Cause: MTU mismatch between nodes in the cluster.

- Fix/Workaround:

  • Reconfigured the MTU settings on all nodes to match the network interface requirements.

  • Rebooted nodes to apply the new MTU settings.

- Lessons Learned:

  Consistent MTU settings across all nodes are crucial for stable networking.

- How to Avoid:

  • Ensure that the MTU configuration is uniform across all cluster nodes.

  • Regularly monitor and verify MTU settings during upgrades.

Scenario #115: Service Load Balancer Failing to Route Traffic to New Pods

- Category: Networking

- Environment: K8s v1.22, Google GKE

  Scenario

- Summary: Service load balancer failed to route traffic to new pods after scaling up.

- What Happened:After scaling up the application pods, the load balancer continued to route traffic to old, terminated pods.

- Diagnosis Steps:

  • Verified pod readiness using kubectl get pods and found that new pods were marked as ready.

  • Inspected the load balancer configuration and found it was not properly refreshing its backend pool.

Root Cause: The service’s load balancer backend pool wasn’t updated when the new pods were created.

- Fix/Workaround:

• Manually refreshed the load balancer’s backend pool configuration.

• Monitored the traffic routing to ensure that it was properly balanced across all pods.

- Lessons Learned:

Load balancer backends need to be automatically updated with new pods.

- How to Avoid:

• Configure the load balancer to auto-refresh backend pools on pod changes.

• Use health checks to ensure only healthy pods are routed traffic.

Scenario #116: Network Traffic Drop Due to Overlapping CIDR Blocks

- Category: Networking

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: Network traffic dropped due to overlapping CIDR blocks between the VPC and Kubernetes pod network.

- What Happened:Overlapping IP ranges between the VPC and pod network caused routing issues and dropped traffic between pods and external services.

- Diagnosis Steps:

  • Reviewed the network configuration and identified the overlap in CIDR blocks.

  • Used kubectl get pods -o wide to inspect pod IPs and found overlaps with the VPC CIDR block.

  Root Cause: Incorrect CIDR block configuration during the cluster setup.

- Fix/Workaround:

  • Reconfigured the pod network CIDR block to avoid overlap with the VPC.

  • Re-deployed the affected pods and confirmed that traffic flow resumed.

- Lessons Learned:

  Plan CIDR block allocations carefully to avoid conflicts.

- How to Avoid:

  • Plan IP address allocations for both the VPC and Kubernetes network in advance.

  • Double-check CIDR blocks during the cluster setup phase.

Scenario #117: Misconfigured DNS Resolvers Leading to Service Discovery Failure

- Category: Networking

- Environment: K8s v1.21, DigitalOcean Kubernetes

Scenario

- Summary: Service discovery failed due to misconfigured DNS resolvers.

- What Happened:A misconfigured DNS resolver in the CoreDNS configuration caused service discovery to fail for some internal services.

- Diagnosis Steps:

  • Checked CoreDNS logs and found that it was unable to resolve certain internal services.

  • Verified that the DNS resolver settings were pointing to incorrect upstream DNS servers.

Root Cause: Incorrect DNS resolver configuration in the CoreDNS config map.

- Fix/Workaround:

  • Corrected the DNS resolver settings in the CoreDNS configuration.

  • Re-applied the configuration and verified that service discovery was restored.

- Lessons Learned:

Always validate DNS resolver configurations during cluster setup.

- How to Avoid:

  • Use default DNS settings if unsure about custom resolver configurations.

  • Regularly verify DNS functionality within the cluster.

Scenario #118: Intermittent Latency Due to Overloaded Network Interface

- Category: Networking

- Environment: K8s v1.22, AWS EKS

  Scenario

- Summary: Intermittent network latency occurred due to an overloaded network interface on a single node.

- What Happened:One node had high network traffic and was not able to handle the load, causing latency spikes.

- Diagnosis Steps:

  • Checked node resource utilization and identified that the network interface was saturated.

  • Verified that the traffic was not being distributed evenly across the nodes.

  Root Cause: Imbalanced network traffic distribution across the node pool.

- Fix/Workaround:

  • Rebalanced the pod distribution across nodes to reduce load on the overloaded network interface.

  • Increased network interface resources on the affected node.

- Lessons Learned:

Proper traffic distribution is key to maintaining low latency.

- How to Avoid:

  • Use autoscaling to dynamically adjust the number of nodes based on traffic load.
  • Monitor network interface usage closely and optimize traffic distribution.

Scenario #119: Pod Disconnection During Network Partition

- Category: Networking

- Environment: K8s v1.20, Google GKE

Scenario

- Summary: Pods were disconnected during a network partition between nodes in the cluster.

- What Happened:A temporary network partition between nodes led to pods becoming disconnected from other services.

- Diagnosis Steps:

  • Used kubectl get events to identify the network partition event.
  • Checked network logs and found that the partition was caused by a temporary routing failure.

Root Cause: Network partition caused pods to lose communication with the rest of the cluster.

- Fix/Workaround:

• Re-established network connectivity and ensured all nodes could communicate with each other.

• Re-scheduled the disconnected pods to different nodes to restore connectivity.

- Lessons Learned:

Network partitioning can cause severe communication issues between pods.

- How to Avoid:

• Use redundant network paths and monitor network stability.

• Enable pod disruption budgets to ensure availability during network issues.

Scenario #121: Pod-to-Pod Communication Blocked by Network Policies

- Category: Networking

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Pod-to-pod communication was blocked due to overly restrictive network policies.

- What Happened:A network policy was misconfigured, preventing certain pods from communicating with each other despite being within the same namespace.

- Diagnosis Steps:

  • Used kubectl get networkpolicy to inspect the network policies in place.

  • Found that a policy restricted traffic between pods in the same namespace.

  • Reviewed policy rules and discovered an incorrect egress restriction.

  Root Cause: Misconfigured egress rule in the network policy.

- Fix/Workaround:

  • Modified the network policy to allow traffic between the pods.

  • Applied the updated policy and verified that communication was restored.

- Lessons Learned:

  Ensure network policies are tested thoroughly before being deployed in production.

- How to Avoid:

  • Use dry-run functionality when applying network policies.

  • Continuously test policies in a staging environment before production rollout.

Scenario #122: Unresponsive External API Due to DNS Resolution Failure

- Category: Networking

- Environment: K8s v1.22, DigitalOcean Kubernetes

Scenario

- Summary: External API calls from the pods failed due to DNS resolution issues for the external domain.

- What Happened:DNS queries for an external API failed due to an incorrect DNS configuration in CoreDNS.

- Diagnosis Steps:

  • Checked CoreDNS logs and found that DNS queries for the external API domain were timing out.

  • Used nslookup to check DNS resolution and found that the query was being routed incorrectly.

Root Cause: Misconfigured upstream DNS server in the CoreDNS configuration.

- Fix/Workaround:

  • Corrected the upstream DNS server settings in CoreDNS.

  • Restarted CoreDNS pods to apply the new configuration.

- Lessons Learned:

Proper DNS resolution setup is critical for communication with external APIs.

- How to Avoid:

  • Regularly monitor CoreDNS health and ensure DNS settings are correctly configured.

  • Use automated health checks to detect DNS issues early.

Scenario #123: Load Balancer Health Checks Failing After Pod Update

- Category: Networking

- Environment: K8s v1.19, GCP Kubernetes Engine

Scenario

- Summary: Load balancer health checks failed after updating a pod due to incorrect readiness probe configuration.

- What Happened:After deploying a new version of the application, the load balancer's health checks started failing, causing traffic to be routed to unhealthy pods.

- Diagnosis Steps:

• Reviewed the load balancer logs and observed failed health checks on newly deployed pods.

• Inspected the pod's readiness probe and found that it was configured incorrectly, leading to premature success.

Root Cause: Incorrect readiness probe causing the pod to be marked healthy before it was ready to serve traffic.

- Fix/Workaround:

• Corrected the readiness probe configuration to reflect the actual application startup time.

• Redeployed the updated pods and verified that they passed the health checks.

- Lessons Learned:

  Always validate readiness probes after updates to avoid traffic disruption.

- How to Avoid:

  • Test readiness probes extensively during staging before updating production.

  • Implement rolling updates to avoid downtime during pod updates.

Scenario #124: Pod Network Performance Degradation After Node Upgrade

- Category: Networking

- Environment: K8s v1.21, Azure AKS

  Scenario

- Summary: Network performance degraded after an automatic node upgrade, causing latency in pod communication.

- What Happened:After an upgrade to a node pool, there was significant latency in network communication between pods, impacting application performance.

- Diagnosis Steps:

  • Checked pod network latency using ping and found increased latency between pods.

  • Examined node and CNI logs, identifying an issue with the upgraded network interface drivers.

  Root Cause: Incompatible network interface drivers following the node upgrade.

- Fix/Workaround:

• Rolled back the node upgrade and manually updated the network interface drivers on the nodes.

• Verified that network performance improved after driver updates.

- Lessons Learned:

Be cautious when performing automatic upgrades in production environments.

- How to Avoid:

• Manually test upgrades in a staging environment before applying them to production.

• Ensure compatibility of network drivers with the Kubernetes version being used.

Scenario #125: Service IP Conflict Due to CIDR Overlap

- Category: Networking

- Environment: K8s v1.20, GKE

Scenario

- Summary: A service IP conflict occurred due to overlapping CIDR blocks, preventing correct routing of traffic to the service.

- What Happened:A new service was assigned an IP within a CIDR range already in use by another service, causing traffic to be routed incorrectly.

- Diagnosis Steps:

• Used kubectl get svc to check the assigned service IPs.

• Noticed the overlapping IP range between the two services.

Root Cause: Overlap in CIDR blocks for services in the same network.

- Fix/Workaround:

• Reconfigured the service CIDR range to avoid conflicts.

• Redeployed services with new IP assignments.

- Lessons Learned:

Plan service CIDR allocations carefully to avoid conflicts.

- How to Avoid:

• Use a dedicated service CIDR block to ensure that IPs are allocated without overlap.

• Automate IP range checks before service creation.

Scenario #126: High Latency in Inter-Namespace Communication

- Category: Networking

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: High latency observed in inter-namespace communication, leading to application timeouts.

- What Happened:Pods in different namespaces experienced significant latency while trying to communicate, causing service timeouts.

- Diagnosis Steps:

• Monitored network latency with kubectl and found inter-namespace traffic was unusually slow.

• Checked network policies and discovered that overly restrictive policies were limiting traffic flow between namespaces.

Root Cause: Overly restrictive network policies blocking inter-namespace traffic.

- Fix/Workaround:

• Modified network policies to allow traffic between namespaces.

• Verified that latency reduced after policy changes.

- Lessons Learned:

Over-restrictive policies can cause performance issues.

- How to Avoid:

• Apply network policies with careful consideration of cross-namespace communication needs.

• Regularly review and update network policies.

Scenario #127: Pod Network Disruptions Due to CNI Plugin Update

- Category: Networking

- Environment: K8s v1.19, DigitalOcean Kubernetes

  Scenario

- Summary: Pods experienced network disruptions after updating the CNI plugin to a newer version.

- What Happened:After upgrading the CNI plugin, network connectivity between pods was disrupted, causing intermittent traffic drops.

- Diagnosis Steps:

  • Checked CNI plugin logs and found that the new version introduced a bug affecting pod networking.

  • Downgraded the CNI plugin version to verify that the issue was related to the upgrade.

  Root Cause: A bug in the newly installed version of the CNI plugin.

- Fix/Workaround:

  • Rolled back to the previous version of the CNI plugin.

  • Reported the bug to the plugin maintainers and kept the older version in place until a fix was released.

- Lessons Learned:

Always test new CNI plugin versions in a staging environment before upgrading production clusters.

- How to Avoid:

• Implement a thorough testing procedure for CNI plugin upgrades.

• Use version locking for CNI plugins to avoid unintentional upgrades.

Scenario #128: Loss of Service Traffic Due to Missing Ingress Annotations

- Category: Networking

- Environment: K8s v1.21, GKE

Scenario

- Summary: Loss of service traffic after ingress annotations were incorrectly set, causing the ingress controller to misroute traffic.

- What Happened:A misconfiguration in the ingress annotations caused the ingress controller to fail to route external traffic to the correct service.

- Diagnosis Steps:

• Inspected ingress resource annotations and found missing or incorrect annotations for the ingress controller.

• Corrected the annotations and re-applied the ingress configuration.

Root Cause: Incorrect ingress annotations caused routing failures.

- Fix/Workaround:

  • Fixed the ingress annotations and re-deployed the ingress resource.
  • Verified traffic flow from external sources to the service was restored.

- Lessons Learned:
  Ensure that ingress annotations are correctly specified for the ingress controller in use.

- How to Avoid:

  • Double-check ingress annotations before applying them to production.
  • Automate ingress validation as part of the CI/CD pipeline.

Scenario #129: Node Pool Draining Timeout Due to Slow Pod Termination

- Category: Cluster Management

- Environment: K8s v1.19, GKE
  Scenario

- Summary: The node pool draining process timed out during upgrades due to pods taking longer than expected to terminate.

- What Happened:During a node pool upgrade, the nodes took longer to drain due to some pods having long graceful termination periods. This caused the upgrade process to time out.

- Diagnosis Steps:

  • Observed that kubectl get pods showed several pods in the terminating state for extended periods.

  • Checked pod logs and noted that they were waiting for a cleanup process to complete during termination.

  Root Cause: Slow pod termination due to resource cleanup tasks caused delays in the node draining process.

- Fix/Workaround:

  • Reduced the grace period for pod termination.

  • Optimized resource cleanup tasks in the pods to reduce termination times.

- Lessons Learned:

  Pod termination times should be minimized to avoid delays during node drains or upgrades.

- How to Avoid:

  • Optimize pod termination logic and cleanup tasks to ensure quicker pod termination.

  • Regularly test node draining during cluster maintenance to identify potential issues.

Scenario #130: Failed Cluster Upgrade Due to Incompatible API Versions

- Category: Cluster Management

- Environment: K8s v1.17, Azure AKS

Scenario

- Summary: The cluster upgrade failed because certain deprecated API versions were still in use, causing compatibility issues with the new K8s version.

- What Happened:The upgrade to K8s v1.18 was blocked due to deprecated API versions still being used in certain resources, such as extensions/v1beta1 for Ingress and ReplicaSets.

- Diagnosis Steps:

• Checked the upgrade logs and identified that the upgrade failed due to the use of deprecated API versions.

• Inspected Kubernetes manifests for resources still using deprecated APIs and discovered several resources in the cluster using old API versions.

Root Cause: The use of deprecated API versions prevented the upgrade to a newer Kubernetes version.

- Fix/Workaround:

• Updated Kubernetes manifests to use the latest stable API versions.

• Re-applied the updated resources and retried the cluster upgrade.

- Lessons Learned:

Always update API versions to ensure compatibility with new Kubernetes versions before performing upgrades.

- How to Avoid:

• Regularly audit API versions in use across the cluster.

• Use tools like kubectl deprecations or kubectl check to identify deprecated resources before upgrades.

Scenario #131: DNS Resolution Failure for Services After Pod Restart

- Category: Networking

- Environment: K8s v1.19, Azure AKS

Scenario

- Summary: DNS resolution failed for services after restarting a pod, causing internal communication issues.

- What Happened:After restarting a pod, the DNS resolution failed for internal services, preventing communication between dependent services.

- Diagnosis Steps:

• Checked CoreDNS logs and found that the pod's DNS cache was stale.

• Verified that the DNS server address was correctly configured in the pod’s /etc/resolv.conf.

Root Cause: DNS cache not properly refreshed after pod restart.

- Fix/Workaround:

  • Restarted CoreDNS to clear the stale cache.

  • Verified that DNS resolution worked for services after the cache refresh.

- Lessons Learned:

  Ensure that DNS caches are cleared or refreshed when a pod restarts.

- How to Avoid:

  • Monitor DNS resolution and configure automatic cache refreshing.

  • Validate DNS functionality after pod restarts.

Scenario #132: Pod IP Address Changes Causing Application Failures

- Category: Networking

- Environment: K8s v1.21, GKE

  Scenario

- Summary: Application failed after a pod IP address changed unexpectedly, breaking communication between services.

- What Happened:The application relied on static pod IPs, but after a pod was rescheduled, its IP address changed, causing communication breakdowns.

- Diagnosis Steps:

• Checked pod logs and discovered that the application failed to reconnect after the IP change.

• Verified that the application was using static pod IPs instead of service names for communication.

Root Cause: Hardcoded pod IPs in the application configuration.

- Fix/Workaround:

• Updated the application to use service DNS names instead of pod IPs.

• Redeployed the application with the new configuration.

- Lessons Learned:

Avoid using static pod IPs in application configurations.

- How to Avoid:

• Use Kubernetes service names to ensure stable communication.

• Set up proper service discovery mechanisms within applications.

Scenario #133: Service Exposure Failed Due to Misconfigured Load Balancer

- Category: Networking

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: A service exposure attempt failed due to incorrect configuration of the AWS load balancer.

- What Happened:The AWS load balancer was misconfigured, resulting in no traffic being routed to the service.

- Diagnosis Steps:

  • Checked the service type (LoadBalancer) and AWS load balancer logs.

  • Found that security group rules were preventing traffic from reaching the service.

  Root Cause: Incorrect security group configuration for the load balancer.

- Fix/Workaround:

  • Modified the security group rules to allow traffic on the necessary ports.

  • Re-deployed the service with the updated configuration.

- Lessons Learned:

  Always review and verify security group rules when using load balancers.

- How to Avoid:

  • Automate security group configuration checks.

  • Implement a robust testing process for load balancer configurations.

Scenario #134: Network Latency Spikes During Pod Autoscaling

- Category: Networking

- Environment: K8s v1.20, Google Cloud

Scenario

- Summary: Network latency spikes occurred when autoscaling pods during traffic surges.

- What Happened:As the number of pods increased due to autoscaling, network latency between pods and services spiked, causing slow response times.

- Diagnosis Steps:

• Monitored pod-to-pod network latency using kubectl and found high latencies during autoscaling events.

• Investigated pod distribution and found that new pods were being scheduled on nodes with insufficient network capacity.

Root Cause: Insufficient network capacity on newly provisioned nodes during autoscaling.

- Fix/Workaround:

• Adjusted the autoscaling configuration to ensure new pods are distributed across nodes with better network resources.

• Increased network capacity for nodes with higher pod density.

- Lessons Learned:

Network resources should be a consideration when autoscaling pods.

- How to Avoid:

• Use network resource metrics to guide autoscaling decisions.

• Continuously monitor and adjust network resources for autoscaling scenarios.

Scenario #135: Service Not Accessible Due to Incorrect Namespace Selector

- Category: Networking

- Environment: K8s v1.18, on-premise

Scenario

- Summary: A service was not accessible due to a misconfigured namespace selector in the service definition.

- What Happened:The service had a namespaceSelector field configured incorrectly, which caused it to be inaccessible from the intended namespace.

- Diagnosis Steps:

• Inspected the service definition and found that the namespaceSelector was set to an incorrect value.

• Verified the intended namespace and adjusted the selector.

Root Cause: Incorrect namespace selector configuration in the service.

- Fix/Workaround:

• Corrected the namespace selector in the service definition.

• Redeployed the service to apply the fix.

- Lessons Learned:

Always carefully validate service selectors, especially when involving namespaces.

- How to Avoid:

• Regularly audit service definitions for misconfigurations.

• Implement automated validation checks for Kubernetes resources.

Scenario #136: Intermittent Pod Connectivity Due to Network Plugin Bug

- Category: Networking

- Environment: K8s v1.23, DigitalOcean Kubernetes

Scenario

- Summary: Pods experienced intermittent connectivity issues due to a bug in the CNI network plugin.

- What Happened:After a network plugin upgrade, some pods lost network connectivity intermittently, affecting communication with other services.

- Diagnosis Steps:

• Checked CNI plugin logs and found errors related to pod IP assignment.

• Rolled back the plugin version and tested connectivity, which resolved the issue.

Root Cause: Bug in the newly deployed version of the CNI plugin.

- Fix/Workaround:

  • Rolled back the CNI plugin to the previous stable version.

  • Reported the bug to the plugin maintainers for a fix.

- Lessons Learned:

  Always test new plugin versions in a staging environment before upgrading in production.

- How to Avoid:

  • Use a canary deployment strategy for CNI plugin updates.

  • Monitor pod connectivity closely after updates.

Scenario #137: Failed Ingress Traffic Routing Due to Missing Annotations

- Category: Networking

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Ingress traffic was not properly routed to services due to missing annotations in the ingress resource.

- What Happened:A missing annotation caused the ingress controller to not route external traffic to the right service.

- Diagnosis Steps:

 • Inspected the ingress resource and found missing or incorrect annotations required for routing traffic correctly.

 • Applied the correct annotations to the ingress resource.

 Root Cause: Missing ingress controller-specific annotations.

- Fix/Workaround:

 • Added the correct annotations to the ingress resource.

 • Redeployed the ingress resource and confirmed traffic routing was restored.

- Lessons Learned:

 Always verify the required annotations for the ingress controller.

- How to Avoid:

 • Use a standard template for ingress resources.

 • Automate the validation of ingress configurations before applying them.

Scenario #138: Pod IP Conflict Causing Service Downtime

- Category: Networking

- Environment: K8s v1.19, GKE

Scenario

- Summary: A pod IP conflict caused service downtime and application crashes.

- What Happened:Two pods were assigned the same IP address by the CNI plugin, leading to network issues and service downtime.

- Diagnosis Steps:

  • Investigated pod IP allocation and found a conflict between two pods.

  • Checked CNI plugin logs and discovered a bug in IP allocation logic.

Root Cause: CNI plugin bug causing duplicate pod IPs.

- Fix/Workaround:

  • Restarted the affected pods, which resolved the IP conflict.

  • Reported the issue to the CNI plugin developers and applied a bug fix.

- Lessons Learned:

Avoid relying on automatic IP allocation without proper checks.

- How to Avoid:

  • Use a custom IP range and monitoring for pod IP allocation.

  • Stay updated with CNI plugin releases and known bugs.

Scenario #139: Latency Due to Unoptimized Service Mesh Configuration

- Category: Networking

- Environment: K8s v1.21, Istio

  Scenario

- Summary: Increased latency in service-to-service communication due to suboptimal configuration of Istio service mesh.

- What Happened:Service latency increased because the Istio service mesh was not optimized for production traffic.

- Diagnosis Steps:

  • Checked Istio configuration for service mesh routing policies.

  • Found that default retry settings were causing unnecessary overhead.

  Root Cause: Misconfigured Istio retries and timeout settings.

- Fix/Workaround:

  • Optimized Istio retry policies to avoid excessive retries.

  • Adjusted timeouts and circuit breakers for better performance.

- Lessons Learned:

  Properly configure and fine-tune service mesh settings for production environments.

- How to Avoid:

• Regularly review and optimize Istio configurations.

• Use performance benchmarks to guide configuration changes.

Scenario #139: DNS Resolution Failure After Cluster Upgrade

- Category: Networking

- Environment: K8s v1.20 to v1.21, AWS EKS

Scenario

- Summary: DNS resolution failures occurred across pods after a Kubernetes cluster upgrade.

- What Happened:After upgrading the Kubernetes cluster, DNS resolution stopped working for certain namespaces, causing intermittent application failures.

- Diagnosis Steps:

• Checked CoreDNS logs and found no errors, but DNS queries were timing out.

• Verified that the upgrade process had updated the CoreDNS deployment, but the config map was not updated correctly.

Root Cause: Misconfiguration in the CoreDNS config map after the cluster upgrade.

- Fix/Workaround:

• Updated the CoreDNS config map to the correct version.

• Restarted CoreDNS pods to apply the updated config.

- Lessons Learned:

After upgrading the cluster, always validate the configuration of critical components like CoreDNS.

- How to Avoid:

• Automate the validation of key configurations after an upgrade.

• Implement pre-upgrade checks to ensure compatibility with existing configurations.

Scenario #140: Service Mesh Sidecar Injection Failure

- Category: Networking

- Environment: K8s v1.19, Istio 1.8

Scenario

- Summary: Sidecar injection failed for some pods in the service mesh, preventing communication between services.

- What Happened:Newly deployed pods in the service mesh were missing their sidecar proxy containers, causing communication failures.

- Diagnosis Steps:

• Verified the Istio sidecar injector webhook was properly configured.

• Checked the labels and annotations on the affected pods and found that they were missing the sidecar.istio.io/inject: "true" annotation.

Root Cause: Pods were missing the required annotation for automatic sidecar injection.

- Fix/Workaround:

• Added the sidecar.istio.io/inject: "true" annotation to the missing pods.

• Redeployed the pods to trigger sidecar injection.

- Lessons Learned:

Ensure that required annotations are applied to all pods, or configure the sidecar injector to inject by default.

- How to Avoid:

• Automate the application of the sidecar.istio.io/inject annotation.

• Use Helm or operators to manage sidecar injection for consistency.

Scenario #141: Network Bandwidth Saturation During Large-Scale Deployments

- Category: Networking

- Environment: K8s v1.21, Azure AKS

Scenario

- Summary: Network bandwidth was saturated during a large-scale deployment, affecting cluster communication.

- What Happened:During a large-scale application deployment, network traffic consumed all available bandwidth, leading to service timeouts and network packet loss.

- Diagnosis Steps:

  • Monitored network traffic and found that the deployment was causing spikes in bandwidth utilization.

  • Identified large Docker images being pulled and deployed across nodes.

  Root Cause: Network bandwidth saturation caused by the simultaneous pulling of large Docker images.

- Fix/Workaround:

  • Staggered the deployment of pods to distribute the load more evenly.

  • Used a local registry to reduce the impact of external image pulls.

- Lessons Learned:

  Ensure that large-scale deployments are distributed in a way that does not overwhelm the network.

- How to Avoid:

  • Use image caching and local registries for large deployments.

  • Implement deployment strategies to stagger or batch workloads.

Scenario #142: Inconsistent Network Policies Blocking Internal Traffic

- Category: Networking

- Environment: K8s v1.18, GKE

Scenario

- Summary: Internal pod-to-pod traffic was unexpectedly blocked due to inconsistent network policies.

- What Happened:After applying a set of network policies, pods in the same namespace could no longer communicate, even though they should have been allowed by the policy.

- Diagnosis Steps:

• Reviewed the network policies and found conflicting ingress rules between services.

• Analyzed logs of the blocked pods and confirmed that network traffic was being denied due to incorrect policy definitions.

Root Cause: Conflicting network policy rules that denied internal traffic.

- Fix/Workaround:

• Merged conflicting network policy rules to allow the necessary traffic.

• Applied the corrected policy and verified that pod communication was restored.

- Lessons Learned:

Network policies need careful management to avoid conflicting rules that can block internal communication.

- How to Avoid:

• Implement a policy review process before applying network policies to production environments.

• Use tools like Calico to visualize and validate network policies before deployment.

Scenario #143: Pod Network Latency Caused by Overloaded CNI Plugin

- Category: Networking

- Environment: K8s v1.19, on-premise

Scenario

- Summary: Pod network latency increased due to an overloaded CNI plugin.

- What Happened:Network latency increased across pods as the CNI plugin (Flannel) became overloaded with traffic, causing service degradation.

- Diagnosis Steps:

• Monitored CNI plugin performance and found high CPU usage due to excessive traffic handling.

• Verified that the nodes were not running out of resources, but the CNI plugin was overwhelmed.

Root Cause: CNI plugin was not optimized for the high volume of network traffic.

- Fix/Workaround:

  • Switched to a more efficient CNI plugin (Calico) to handle the traffic load.

  • Tuned the Calico settings to optimize performance under heavy load.

- Lessons Learned:

  Always ensure that the CNI plugin is well-suited to the network load expected in production environments.

- How to Avoid:

  • Test and benchmark CNI plugins before deploying in production.

  • Regularly monitor the performance of the CNI plugin and adjust configurations as needed.

Scenario #144: TCP Retransmissions Due to Network Saturation

- Category: Networking

- Environment: K8s v1.22, DigitalOcean Kubernetes

  Scenario

- Summary: TCP retransmissions increased due to network saturation, leading to degraded pod-to-pod communication.

- What Happened:Pods in the cluster started experiencing increased latency and timeouts, which was traced back to TCP retransmissions caused by network saturation.

- Diagnosis Steps:

• Analyzed network performance using tcpdump and found retransmissions occurring during periods of high traffic.

• Verified that there was no hardware failure, but network bandwidth was fully utilized.

Root Cause: Insufficient network bandwidth during high traffic periods.

- Fix/Workaround:

• Increased network bandwidth allocation for the cluster.

• Implemented QoS policies to prioritize critical traffic.

- Lessons Learned:

Network saturation can severely affect pod communication, especially under heavy loads.

- How to Avoid:

• Use quality-of-service (QoS) and bandwidth throttling to prevent network saturation.

• Regularly monitor network bandwidth and adjust scaling policies to meet traffic demands.

Scenario #145: DNS Lookup Failures Due to Resource Limits

- Category: Networking

- Environment: K8s v1.20, AWS EKS

Scenario

- Summary: DNS lookup failures occurred due to resource limits on the CoreDNS pods.

- What Happened:CoreDNS pods hit their CPU and memory resource limits, causing DNS queries to fail intermittently.

- Diagnosis Steps:

• Checked CoreDNS logs and identified that it was consistently hitting resource limits.

• Verified that the node resources were underutilized, but CoreDNS had been allocated insufficient resources.

Root Cause: Insufficient resource limits set for CoreDNS pods.

- Fix/Workaround:

• Increased the resource limits for CoreDNS pods to handle the load.

• Restarted the CoreDNS pods to apply the new resource limits.

- Lessons Learned:

Always allocate sufficient resources for critical components like CoreDNS.

- How to Avoid:

• Set resource requests and limits for critical services based on actual usage.

• Use Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale resource allocation for CoreDNS.

Scenario #146: Service Exposure Issues Due to Incorrect Ingress Configuration

- Category: Networking

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: A service was not accessible externally due to incorrect ingress configuration.

- What Happened:External traffic could not access the service because the ingress controller was misconfigured.

- Diagnosis Steps:

• Checked the ingress controller logs and found that the ingress was incorrectly pointing to an outdated service.

• Verified the ingress configuration and discovered a typo in the service URL.

Root Cause: Misconfiguration in the ingress resource that directed traffic to the wrong service.

- Fix/Workaround:

• Corrected the service URL in the ingress resource.

• Redeployed the ingress configuration.

- Lessons Learned:

Ingress configurations need careful attention to detail, especially when specifying service URLs.

- How to Avoid:

• Use automated testing and validation tools for ingress resources.

• Document standard ingress configurations to avoid errors.

Scenario #147: Pod-to-Pod Communication Failure Due to Network Policy

- Category: Networking

- Environment: K8s v1.19, on-premise

Scenario

- Summary: Pod-to-pod communication failed due to an overly restrictive network policy.

- What Happened:Pods in the same namespace could not communicate because an ingress network policy blocked traffic between them.

- Diagnosis Steps:

• Examined network policies and identified that the ingress policy was too restrictive.

• Verified pod logs and found that traffic was being denied by the network policy.

Root Cause: Overly restrictive network policy that blocked pod-to-pod communication.

- Fix/Workaround:

• Updated the network policy to allow traffic between pods in the same namespace.

• Applied the updated policy and verified that communication was restored.

- Lessons Learned:

Carefully review network policies to ensure they do not unintentionally block necessary traffic.

- How to Avoid:

• Use a policy auditing tool to ensure network policies are properly defined and do not block essential traffic.

• Regularly test network policies in staging environments.

Scenario #148: Unstable Network Due to Overlay Network Misconfiguration

- Category: Networking

- Environment: K8s v1.18, VMware Tanzu

Scenario

- Summary: The overlay network was misconfigured, leading to instability in pod communication.

- What Happened:After deploying an application, pod communication became unstable due to misconfiguration in the overlay network.

- Diagnosis Steps:

  • Reviewed the CNI plugin (Calico) logs and found incorrect IP pool configurations.

  • Identified that the overlay network was not providing consistent routing between pods.

  Root Cause: Incorrect overlay network configuration.

- Fix/Workaround:

  • Corrected the IP pool configuration in the Calico settings.

  • Restarted Calico pods to apply the fix.

- Lessons Learned:

  Carefully validate overlay network configurations to ensure proper routing and stability.

- How to Avoid:

  • Test network configurations in staging environments before deploying to production.

  • Regularly audit network configurations for consistency.

Scenario #149: Intermittent Pod Network Connectivity Due to Cloud Provider Issues

- Category: Networking

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Pod network connectivity was intermittent due to issues with the cloud provider's network infrastructure.

- What Happened:Pods experienced intermittent network connectivity, and communication between nodes was unreliable.

- Diagnosis Steps:

• Used AWS CloudWatch to monitor network metrics and identified sporadic outages in the cloud provider’s network infrastructure.

• Verified that the Kubernetes network infrastructure was working correctly.

Root Cause: Cloud provider network outages affecting pod-to-pod communication.

- Fix/Workaround:

• Waited for the cloud provider to resolve the network issue.

• Implemented automatic retries in application code to mitigate the impact of intermittent connectivity.

- Lessons Learned:

Be prepared for cloud provider network outages and implement fallback mechanisms.

- How to Avoid:

• Set up alerts for cloud provider outages and implement retries in critical network-dependent applications.

• Design applications to be resilient to network instability.

Scenario #150: Port Conflicts Between Services in Different Namespaces

- Category: Networking

- Environment: K8s v1.22, Google GKE

Scenario

- Summary: Port conflicts between services in different namespaces led to communication failures.

- What Happened:Two services in different namespaces were configured to use the same port number, causing a conflict in service communication.

- Diagnosis Steps:

• Checked service configurations and found that both services were set to expose port 80.

• Verified pod logs and found that traffic to one service was being routed to another due to the port conflict.

Root Cause: Port conflicts between services in different namespaces.

- Fix/Workaround:

• Updated the service definitions to use different ports for the conflicting services.

• Redeployed the services and verified communication.

- Lessons Learned:

Avoid port conflicts by ensuring that services in different namespaces use unique ports.

- How to Avoid:

• Use unique port allocations across services in different namespaces.

• Implement service naming conventions that include port information.

Scenario #151: NodePort Service Not Accessible Due to Firewall Rules

- Category: Networking

- Environment: K8s v1.23, Google GKE

Scenario

- Summary: A NodePort service became inaccessible due to restrictive firewall rules on the cloud provider.

- What Happened:External access to a service using a NodePort was blocked because the cloud provider's firewall rules were too restrictive.

- Diagnosis Steps:

• Checked service configuration and confirmed that it was correctly exposed as a NodePort.

• Used kubectl describe svc to verify the NodePort assigned.

• Verified the firewall rules for the cloud provider and found that ingress was blocked on the NodePort range.

Root Cause: Firewall rules on the cloud provider were not configured to allow traffic on the NodePort range.

- Fix/Workaround:

• Updated the firewall rules to allow inbound traffic to the NodePort range.

• Ensured that the required port was open on all nodes.

- Lessons Learned:

Always check cloud firewall rules when exposing services using NodePort.

- How to Avoid:

• Automate the validation of firewall rules after deploying NodePort services.

• Document and standardize firewall configurations for all exposed services.

Scenario #152: DNS Latency Due to Overloaded CoreDNS Pods

- Category: Networking

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: CoreDNS latency increased due to resource constraints on the CoreDNS pods.

- What Happened:CoreDNS started experiencing high response times due to CPU and memory resource constraints, leading to DNS resolution delays.

- Diagnosis Steps:

  • Checked CoreDNS pod resource usage and found high CPU usage.

  • Verified that DNS resolution was slowing down for multiple namespaces and services.

  • Increased logging verbosity for CoreDNS and identified high query volume.

Root Cause: CoreDNS pods did not have sufficient resources allocated to handle the query load.

- Fix/Workaround:

  • Increased CPU and memory resource limits for CoreDNS pods.

  • Restarted CoreDNS pods to apply the new resource limits.

- Lessons Learned:

CoreDNS should be allocated appropriate resources based on expected load, especially in large clusters.

- How to Avoid:

  • Set resource requests and limits for CoreDNS based on historical query volume.

• Monitor CoreDNS performance and scale resources dynamically.

Scenario #153: Network Performance Degradation Due to Misconfigured MTU

- Category: Networking

- Environment: K8s v1.20, on-premise

Scenario

- Summary: Network performance degraded due to an incorrect Maximum Transmission Unit (MTU) setting.

- What Happened:Network performance between pods degraded after a change in the MTU settings in the CNI plugin.

- Diagnosis Steps:

• Used ping tests to diagnose high latency and packet drops between nodes.

• Verified MTU settings on the nodes and CNI plugin, and found that the MTU was mismatched between the nodes and the CNI.

Root Cause: MTU mismatch between Kubernetes nodes and the CNI plugin.

- Fix/Workaround:

• Aligned the MTU settings between the CNI plugin and the Kubernetes nodes.

• Rebooted affected nodes to apply the configuration changes.

- Lessons Learned:

Ensure that MTU settings are consistent across the network stack to avoid performance degradation.

- How to Avoid:

• Implement monitoring and alerting for MTU mismatches.

• Validate network configurations before applying changes to the CNI plugin.

Scenario #154: Application Traffic Routing Issue Due to Incorrect Ingress Resource

- Category: Networking

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: Application traffic was routed incorrectly due to an error in the ingress resource definition.

- What Happened:Traffic intended for a specific application was routed to the wrong backend service because the ingress resource had a misconfigured path.

- Diagnosis Steps:

• Reviewed the ingress resource and found that the path definition did not match the expected URL.

• Validated that the backend service was correctly exposed and running.

Root Cause: Incorrect path specification in the ingress resource, causing traffic to be routed incorrectly.

- Fix/Workaround:

• Corrected the path definition in the ingress resource.

• Redeployed the ingress configuration to ensure correct traffic routing.

- Lessons Learned:

Always carefully review and test ingress path definitions before applying them in production.

- How to Avoid:

• Implement a staging environment to test ingress resources before production deployment.

• Use automated tests to verify ingress configuration correctness.

Scenario #155: Intermittent Service Disruptions Due to DNS Caching Issue

- Category: Networking

- Environment: K8s v1.21, GCP GKE

Scenario

- Summary: Intermittent service disruptions occurred due to stale DNS cache in CoreDNS.

- What Happened:Services failed intermittently because CoreDNS had cached stale DNS records, causing them to resolve incorrectly.

- Diagnosis Steps:

  • Verified DNS resolution using nslookup and found incorrect IP addresses being returned.

  • Cleared the DNS cache in CoreDNS and noticed that the issue was temporarily resolved.

  Root Cause: CoreDNS was caching stale DNS records due to incorrect TTL settings.

- Fix/Workaround:

  • Reduced the TTL value in CoreDNS configuration.

  • Restarted CoreDNS pods to apply the new TTL setting.

- Lessons Learned:

  Be cautious of DNS TTL settings, especially in dynamic environments where IP addresses change frequently.

- How to Avoid:

  • Monitor DNS records and TTL values actively.

  • Implement cache invalidation or reduce TTL for critical services.

Scenario #156: Flannel Overlay Network Interruption Due to Node Failure

- Category: Networking

- Environment: K8s v1.18, on-premise

Scenario

- Summary: Flannel overlay network was interrupted after a node failure, causing pod-to-pod communication issues.

- What Happened:A node failure caused the Flannel CNI plugin to lose its network routes, disrupting communication between pods on different nodes.

- Diagnosis Steps:

• Used kubectl get pods -o wide to identify affected pods.

• Checked the Flannel daemon logs and found errors related to missing network routes.

Root Cause: Flannel CNI plugin was not re-establishing network routes after the node failure.

- Fix/Workaround:

• Restarted the Flannel pods on the affected nodes to re-establish network routes.

• Verified that communication between pods was restored.

- Lessons Learned:

Ensure that CNI plugins can gracefully handle node failures and re-establish connectivity.

- How to Avoid:

  • Implement automatic recovery or self-healing mechanisms for CNI plugins.

  • Monitor CNI plugin logs to detect issues early.

Scenario #157: Network Traffic Loss Due to Port Collision in Network Policy

- Category: Networking

- Environment: K8s v1.19, GKE

  Scenario

- Summary: Network traffic was lost due to a port collision in the network policy, affecting application availability.

- What Happened:Network traffic was dropped because a network policy inadvertently blocked traffic to a port that was required by another application.

- Diagnosis Steps:

  • Inspected the network policy using kubectl describe netpol and identified the port conflict.

  • Verified traffic flow using kubectl logs to identify blocked traffic.

  Root Cause: Misconfigured network policy that blocked traffic to a necessary port due to port collision.

- Fix/Workaround:

• Updated the network policy to allow the necessary port.

• Applied the updated network policy and tested the traffic flow.

- Lessons Learned:

Thoroughly test network policies to ensure that they do not block critical application traffic.

- How to Avoid:

• Review network policies in detail before applying them in production.

• Use automated tools to validate network policies.

Scenario #158: CoreDNS Service Failures Due to Resource Exhaustion

- Category: Networking

- Environment: K8s v1.20, Azure AKS

Scenario

- Summary: CoreDNS service failed due to resource exhaustion, causing DNS resolution failures.

- What Happened:CoreDNS pods exhausted available CPU and memory, leading to service failures and DNS resolution issues.

- Diagnosis Steps:

• Checked CoreDNS logs and found out-of-memory errors.

• Verified that the CPU usage was consistently high for the CoreDNS pods.

Root Cause: Insufficient resources allocated to CoreDNS pods, causing service crashes.

- Fix/Workaround:

• Increased the resource requests and limits for CoreDNS pods.

• Restarted the CoreDNS pods to apply the updated resource allocation.

- Lessons Learned:

Ensure that critical components like CoreDNS have sufficient resources allocated for normal operation.

- How to Avoid:

• Set appropriate resource requests and limits based on usage patterns.

• Monitor resource consumption of CoreDNS and other critical components.

Scenario #159: Pod Network Partition Due to Misconfigured IPAM

- Category: Networking

- Environment: K8s v1.22, VMware Tanzu

Scenario

- Summary: Pod network partition occurred due to an incorrectly configured IP Address Management (IPAM) in the CNI plugin.

- What Happened:Pods were unable to communicate across nodes because the IPAM configuration was improperly set, causing an address space overlap.

- Diagnosis Steps:

  • Inspected the CNI configuration and discovered overlapping IP address ranges.

  • Verified network policies and found no conflicts, but the IP address allocation was incorrect.

  Root Cause: Misconfiguration of IPAM settings in the CNI plugin.

- Fix/Workaround:

  • Corrected the IPAM configuration to use non-overlapping IP address ranges.

  • Redeployed the CNI plugin and restarted affected pods.

- Lessons Learned:

  Carefully configure IPAM in CNI plugins to prevent network address conflicts.

- How to Avoid:

  • Validate network configurations before deploying.

  • Use automated checks to detect IP address conflicts in multi-node environments.

Scenario #160: Network Performance Degradation Due to Overloaded CNI Plugin

- Category: Networking

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Network performance degraded due to the CNI plugin being overwhelmed by high traffic volume.

- What Happened:A sudden spike in traffic caused the CNI plugin to become overloaded, resulting in significant packet loss and network latency between pods.

- Diagnosis Steps:

  • Monitored network traffic using kubectl top pods and observed unusually high traffic to and from a few specific pods.

  • Inspected CNI plugin logs and found errors related to resource exhaustion.

  Root Cause: The CNI plugin lacked sufficient resources to handle the spike in traffic, leading to packet loss and network degradation.

- Fix/Workaround:

  • Increased resource limits for the CNI plugin pods.

  • Used network policies to limit the traffic spikes to specific services.

- Lessons Learned:

  Ensure that the CNI plugin is properly sized to handle peak traffic loads, and monitor its health regularly.

- How to Avoid:

  • Set up traffic rate limiting to prevent sudden spikes from overwhelming the network.

  • Use resource limits and horizontal pod autoscaling for critical CNI components.

Scenario #161: Network Performance Degradation Due to Overloaded CNI Plugin

- Category: Networking

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Network performance degraded due to the CNI plugin being overwhelmed by high traffic volume.

- What Happened:A sudden spike in traffic caused the CNI plugin to become overloaded, resulting in significant packet loss and network latency between pods.

- Diagnosis Steps:

  • Monitored network traffic using kubectl top pods and observed unusually high traffic to and from a few specific pods.

  • Inspected CNI plugin logs and found errors related to resource exhaustion.

  Root Cause: The CNI plugin lacked sufficient resources to handle the spike in traffic, leading to packet loss and network degradation.

- Fix/Workaround:

• Increased resource limits for the CNI plugin pods.

• Used network policies to limit the traffic spikes to specific services.

- Lessons Learned:

Ensure that the CNI plugin is properly sized to handle peak traffic loads, and monitor its health regularly.

- How to Avoid:

• Set up traffic rate limiting to prevent sudden spikes from overwhelming the network.

• Use resource limits and horizontal pod autoscaling for critical CNI components.

Scenario #162: DNS Resolution Failures Due to Misconfigured CoreDNS

- Category: Networking

- Environment: K8s v1.19, Google GKE

Scenario

- Summary: DNS resolution failures due to misconfigured CoreDNS, leading to application errors.

- What Happened:CoreDNS was misconfigured with the wrong upstream DNS resolver, causing DNS lookups to fail and leading to application connectivity issues.

- Diagnosis Steps:

• Ran kubectl logs -l k8s-app=coredns to view the CoreDNS logs and identified errors related to upstream DNS resolution.

• Used kubectl get configmap coredns -n kube-system -o yaml to inspect the CoreDNS configuration.

Root Cause: CoreDNS was configured with an invalid upstream DNS server that was unreachable.

- Fix/Workaround:

• Updated CoreDNS ConfigMap to point to a valid upstream DNS server.

• Restarted CoreDNS pods to apply the new configuration.

- Lessons Learned:

Double-check DNS configurations during deployment and monitor CoreDNS health regularly.

- How to Avoid:

• Automate the validation of DNS configurations and use reliable upstream DNS servers.

• Set up monitoring for DNS resolution latency and errors.

Scenario #163: Network Partition Due to Incorrect Calico Configuration

- Category: Networking

- Environment: K8s v1.20, Azure AKS

Scenario

- Summary: Network partitioning due to incorrect Calico CNI configuration, resulting in pods being unable to communicate with each other.

- What Happened:Calico was misconfigured with an incorrect CIDR range, leading to network partitioning where some pods could not reach other pods in the same cluster.

- Diagnosis Steps:

• Verified pod connectivity using kubectl exec and confirmed network isolation between pods.

• Inspected Calico configuration and discovered the incorrect CIDR range in the calicoctl configuration.

Root Cause: Incorrect CIDR range in the Calico configuration caused pod networking issues.

- Fix/Workaround:

• Updated the Calico CIDR range configuration to match the cluster's networking plan.

• Restarted Calico pods to apply the new configuration and restore network connectivity.

- Lessons Learned:

Ensure that network configurations, especially for CNI plugins, are thoroughly tested before deployment.

- How to Avoid:

• Use automated network validation tools to check for partitioning and misconfigurations.

• Regularly review and update CNI configuration as the cluster grows.

Scenario #164: IP Overlap Leading to Communication Failure Between Pods

- Category: Networking

- Environment: K8s v1.19, On-premise

Scenario

- Summary: Pods failed to communicate due to IP address overlap caused by an incorrect subnet configuration.

- What Happened:The pod network subnet overlapped with another network on the host machine, causing IP address conflicts and preventing communication between pods.

- Diagnosis Steps:

• Verified pod IPs using kubectl get pods -o wide and identified overlapping IPs with host network IPs.

• Checked network configuration on the host and discovered the overlapping subnet.

Root Cause: Incorrect subnet configuration that caused overlapping IP ranges between the Kubernetes pod network and the host network.

- Fix/Workaround:

• Updated the pod network CIDR range to avoid overlapping with host network IPs.

• Restarted the Kubernetes networking components to apply the new configuration.

- Lessons Learned:

Pay careful attention to subnet planning when setting up networking for Kubernetes clusters to avoid conflicts.

- How to Avoid:

• Use a tool to validate network subnets during cluster setup.

• Avoid using overlapping IP ranges when planning pod and host network subnets.

Scenario #165: Pod Network Latency Due to Overloaded Kubernetes Network Interface

- Category: Networking

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Pod network latency increased due to an overloaded network interface on the Kubernetes nodes.

- What Happened:A sudden increase in traffic caused the network interface on the nodes to become overloaded, leading to high network latency between pods and degraded application performance.

- Diagnosis Steps:

• Used kubectl top node to observe network interface metrics and saw high network throughput and packet drops.

• Checked AWS CloudWatch metrics and confirmed that the network interface was approaching its maximum throughput.

Root Cause: The network interface on the nodes was unable to handle the high network traffic due to insufficient capacity.

- Fix/Workaround:

• Increased the network bandwidth for the AWS EC2 instances hosting the Kubernetes nodes.

• Used network policies to limit traffic to critical pods and avoid overwhelming the network interface.

- Lessons Learned:

Ensure that Kubernetes nodes are provisioned with adequate network capacity for expected traffic loads.

- How to Avoid:

• Monitor network traffic and resource utilization at the node level.

• Scale nodes appropriately or use higher-bandwidth instances for high-traffic workloads.

Scenario #166: Intermittent Connectivity Failures Due to Pod DNS Cache Expiry

- Category: Networking

- Environment: K8s v1.22, Google GKE

Scenario

- Summary: Intermittent connectivity failures due to pod DNS cache expiry, leading to failed DNS lookups for external services.

- What Happened:Pods experienced intermittent connectivity failures because the DNS cache expired too quickly, causing DNS lookups to fail for external services.

- Diagnosis Steps:

  • Checked pod logs and observed errors related to DNS lookup failures.

  • Inspected the CoreDNS configuration and identified a low TTL (time-to-live) value for DNS cache.

Root Cause: The DNS TTL was set too low, causing DNS entries to expire before they could be reused.

- Fix/Workaround:

  • Increased the DNS TTL value in the CoreDNS configuration.

  • Restarted CoreDNS pods to apply the new configuration.

- Lessons Learned:

Proper DNS caching settings are critical for maintaining stable connectivity to external services.

- How to Avoid:

• Set appropriate DNS TTL values based on the requirements of your services.

• Regularly monitor DNS performance and adjust TTL settings as needed.

Scenario #167: Flapping Network Connections Due to Misconfigured Network Policies

- Category: Networking

- Environment: K8s v1.20, Azure AKS

Scenario

- Summary: Network connections between pods were intermittently dropping due to misconfigured network policies, causing application instability.

- What Happened:Network policies were incorrectly configured, leading to intermittent drops in network connectivity between pods, especially under load.

- Diagnosis Steps:

• Used kubectl describe networkpolicy to inspect network policies and found overly restrictive ingress rules.

• Verified pod-to-pod communication using kubectl exec and confirmed that traffic was being blocked intermittently.

Root Cause: Misconfigured network policies that were too restrictive, blocking legitimate traffic between pods.

- Fix/Workaround:

• Updated the network policies to allow necessary pod-to-pod communication.

• Tested connectivity to ensure stability after the update.

- Lessons Learned:

Ensure that network policies are tested thoroughly before being enforced, especially in production.

- How to Avoid:

• Use a staged approach for deploying network policies, first applying them to non-critical pods.

• Implement automated tests to validate network policy configurations.

Scenario #168: Cluster Network Downtime Due to CNI Plugin Upgrade

- Category: Networking

- Environment: K8s v1.22, On-premise

Scenario

- Summary: Cluster network downtime occurred during a CNI plugin upgrade, affecting pod-to-pod communication.

- What Happened:During an upgrade to the CNI plugin, the network was temporarily disrupted due to incorrect version compatibility and missing network configurations.

- Diagnosis Steps:

• Inspected pod logs and noticed failed network interfaces after the upgrade.

• Checked CNI plugin version compatibility and identified missing configurations for the new version.

Root Cause: The new version of the CNI plugin required additional configuration settings that were not applied during the upgrade.

- Fix/Workaround:

• Applied the required configuration changes for the new CNI plugin version.

• Restarted affected pods and network components to restore connectivity.

- Lessons Learned:

Always verify compatibility and required configurations before upgrading the CNI plugin.

- How to Avoid:

• Test plugin upgrades in a staging environment to catch compatibility issues.

• Follow a defined upgrade process that includes validation of configurations.

Scenario #169: Inconsistent Pod Network Connectivity in Multi-Region Cluster

- Category: Networking

- Environment: K8s v1.21, GCP

Scenario

- Summary: Pods in a multi-region cluster experienced inconsistent network connectivity between regions due to misconfigured VPC peering.

- What Happened:The VPC peering between two regions was misconfigured, leading to intermittent network connectivity issues between pods in different regions.

- Diagnosis Steps:

• Used kubectl exec to check network latency and packet loss between pods in different regions.

• Inspected VPC peering settings and found that the correct routes were not configured to allow cross-region traffic.

Root Cause: Misconfigured VPC peering between the regions prevented proper routing of network traffic.

- Fix/Workaround:

• Updated VPC peering routes and ensured proper configuration between the regions.

• Tested connectivity after the change to confirm resolution.

- Lessons Learned:

Ensure that all network routing and peering configurations are validated before deploying cross-region clusters.

- How to Avoid:

• Regularly review VPC and peering configurations.

• Use automated network tests to confirm inter-region connectivity.

Scenario #170: Pod Network Partition Due to Network Policy Blocking DNS Requests

- Category: Networking

- Environment: K8s v1.19, Azure AKS

  Scenario

- Summary: Pods were unable to resolve DNS due to a network policy blocking DNS traffic, causing service failures.

- What Happened:A network policy was accidentally configured to block DNS (UDP port 53) traffic between pods, preventing DNS resolution and causing services to fail.

- Diagnosis Steps:

  • Observed that pods were unable to reach external services, and kubectl exec into the pods showed DNS resolution failures.

  • Used kubectl describe networkpolicy and found the DNS traffic was blocked in the policy.

  Root Cause: The network policy accidentally blocked DNS traffic due to misconfigured ingress and egress rules.

- Fix/Workaround:

  • Updated the network policy to allow DNS traffic.

  • Restarted affected pods to ensure they could access DNS again.

- Lessons Learned:

  Always verify that network policies allow necessary traffic, especially for DNS.

- How to Avoid:

  • Regularly test and validate network policies in non-production environments.

  • Set up monitoring for blocked network traffic.

Scenario #171: Network Bottleneck Due to Overutilized Network Interface

- Category: Networking

- Environment: K8s v1.22, AWS EKS

  Scenario

- Summary: Network bottleneck occurred due to overutilization of a single network interface on the worker nodes.

- What Happened:The worker nodes were using a single network interface to handle both pod traffic and node communication. The high volume of pod traffic caused the network interface to become overutilized, resulting in slow communication.

- Diagnosis Steps:

  • Checked the network interface metrics using AWS CloudWatch and found that the interface was nearing its throughput limit.

• Used kubectl top node and observed high network usage on the affected nodes.

Root Cause: The network interface on the worker nodes was not properly partitioned to handle separate types of traffic, leading to resource contention.

- Fix/Workaround:

• Added a second network interface to the worker nodes for pod traffic and node-to-node communication.

• Reconfigured the nodes to distribute traffic across the two interfaces.

- Lessons Learned:

Proper network interface design is crucial for handling high traffic loads and preventing bottlenecks.

- How to Avoid:

• Design network topologies that segregate different types of traffic (e.g., pod traffic, node communication).

• Regularly monitor network utilization and scale resources as needed.

Scenario #172: Network Latency Caused by Overloaded VPN Tunnel

- Category: Networking

- Environment: K8s v1.20, On-premise

Scenario

- Summary: Network latency increased due to an overloaded VPN tunnel between the Kubernetes cluster and an on-premise data center.

- What Happened:The VPN tunnel between the Kubernetes cluster in the cloud and an on-premise data center became overloaded, causing increased latency for communication between services located in the two environments.

- Diagnosis Steps:

  • Used kubectl exec to measure response times between pods and services in the on-premise data center.

  • Monitored VPN tunnel usage and found it was reaching its throughput limits during peak hours.

  Root Cause: The VPN tunnel was not sized correctly to handle the required traffic between the cloud and on-premise environments.

- Fix/Workaround:

  • Upgraded the VPN tunnel to a higher bandwidth option.

  • Optimized the data flow by reducing unnecessary traffic over the tunnel.

- Lessons Learned:

  Ensure that hybrid network connections like VPNs are appropriately sized and optimized for traffic.

- How to Avoid:

  • Test VPN tunnels with real traffic before moving to production.

  • Monitor tunnel utilization and upgrade bandwidth as needed.

Scenario #173: Dropped Network Packets Due to MTU Mismatch

- Category: Networking

- Environment: K8s v1.21, GKE

  Scenario

- Summary: Network packets were dropped due to a mismatch in Maximum Transmission Unit (MTU) settings across different network components.

- What Happened:Pods experienced connectivity issues and packet loss because the MTU settings on the nodes and CNI plugin were inconsistent, causing packets to be fragmented and dropped.

- Diagnosis Steps:

  • Used ping and tracepath tools to identify dropped packets and packet fragmentation.

  • Inspected the CNI plugin and node MTU configurations and found a mismatch.

  Root Cause: Inconsistent MTU settings between the CNI plugin and the Kubernetes nodes caused packet fragmentation and loss.

- Fix/Workaround:

  • Unified MTU settings across all nodes and the CNI plugin configuration.

  • Restarted the network components to apply the changes.

- Lessons Learned:

Ensure consistent MTU settings across the entire networking stack in Kubernetes clusters.

- How to Avoid:

  • Automate MTU validation checks during cluster setup and upgrades.
  • Monitor network packet loss and fragmentation regularly.

Scenario #174: Pod Network Isolation Due to Misconfigured Network Policy

- Category: Networking

- Environment: K8s v1.20, Azure AKS

  Scenario

- Summary: Pods in a specific namespace were unable to communicate due to an incorrectly applied network policy blocking traffic between namespaces.

- What Happened:A network policy was incorrectly configured to block communication between namespaces, leading to service failures and inability to reach certain pods.

- Diagnosis Steps:

  • Used kubectl describe networkpolicy to inspect the policy and confirmed it was overly restrictive.
  • Tested pod-to-pod communication using kubectl exec and verified the isolation.

Root Cause: The network policy was too restrictive and blocked cross-namespace communication.

- Fix/Workaround:

• Updated the network policy to allow traffic between namespaces.

• Restarted affected pods to re-establish communication.

- Lessons Learned:

Always test network policies in a staging environment to avoid unintentional isolation.

- How to Avoid:

• Use a staged approach to apply network policies and validate them before enforcing them in production.

• Implement automated tests for network policy validation.

Scenario #175: Service Discovery Failures Due to CoreDNS Pod Crash

- Category: Networking

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: Service discovery failures occurred when CoreDNS pods crashed due to resource exhaustion, causing DNS resolution issues.

- What Happened:CoreDNS pods crashed due to high CPU utilization caused by excessive DNS queries, which prevented service discovery and caused communication failures.

- Diagnosis Steps:

  • Checked pod logs and observed frequent crashes related to out-of-memory (OOM) errors.

  • Monitored CoreDNS resource utilization and confirmed CPU spikes from DNS queries.

  Root Cause: Resource exhaustion in CoreDNS due to an overload of DNS queries.

- Fix/Workaround:

  • Increased CPU and memory resources for CoreDNS pods.

  • Optimized the DNS query patterns from applications to reduce the load.

- Lessons Learned:

  Ensure that DNS services like CoreDNS are properly resourced and monitored.

- How to Avoid:

  • Set up monitoring for DNS query rates and resource utilization.

  • Scale CoreDNS horizontally to distribute the load.

Scenario #176: Pod DNS Resolution Failure Due to CoreDNS Configuration Issue

- Category: Networking

- Environment: K8s v1.18, On-premise

Scenario

- Summary: DNS resolution failures occurred within pods due to a misconfiguration in the CoreDNS config map.

- What Happened:CoreDNS was misconfigured to not forward DNS queries to external DNS servers, causing pods to fail when resolving services outside the cluster.

- Diagnosis Steps:

• Ran kubectl exec in the affected pods and verified DNS resolution failure.

• Inspected the CoreDNS ConfigMap and found that the forward section was missing the external DNS servers.

Root Cause: CoreDNS was not configured to forward external queries, leading to DNS resolution failure for non-cluster services.

- Fix/Workaround:

• Updated the CoreDNS ConfigMap to add the missing external DNS server configuration.

• Restarted the CoreDNS pods to apply the changes.

- Lessons Learned:

Always review and test DNS configurations in CoreDNS, especially for hybrid clusters.

- How to Avoid:

• Use automated validation tools to check CoreDNS configuration.

• Set up tests for DNS resolution to catch errors before they impact production.

Scenario #177: DNS Latency Due to Overloaded CoreDNS Pods

- Category: Networking

- Environment: K8s v1.19, GKE

Scenario

- Summary: CoreDNS pods experienced high latency and timeouts due to resource overutilization, causing slow DNS resolution for applications.

- What Happened:CoreDNS pods were handling a high volume of DNS requests without sufficient resources, leading to increased latency and timeouts.

- Diagnosis Steps:

• Used kubectl top pod to observe high CPU and memory usage on CoreDNS pods.

• Checked the DNS query logs and saw long response times.

Root Cause: CoreDNS was under-resourced, and the high DNS traffic caused resource contention.

- Fix/Workaround:

• Increased CPU and memory limits for CoreDNS pods.

• Enabled horizontal pod autoscaling to dynamically scale CoreDNS based on traffic.

- Lessons Learned:

Proper resource allocation and autoscaling are critical for maintaining DNS performance under load.

- How to Avoid:

• Set up resource limits and autoscaling for CoreDNS pods.

• Monitor DNS traffic and resource usage regularly to prevent overloads.

Scenario #178: Pod Network Degradation Due to Overlapping CIDR Blocks

- Category: Networking

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Network degradation occurred due to overlapping CIDR blocks between VPCs in a hybrid cloud setup, causing routing issues.

- What Happened:In a hybrid cloud setup, the CIDR blocks of the Kubernetes cluster VPC and the on-premise VPC overlapped, causing routing issues that led to network degradation and service disruptions.

- Diagnosis Steps:

• Investigated network routes using kubectl describe node and confirmed overlapping CIDR blocks.

• Verified routing tables and identified conflicts causing packets to be misrouted.

Root Cause: Overlapping CIDR blocks between the cluster VPC and the on-premise VPC caused routing conflicts.

- Fix/Workaround:

• Reconfigured the CIDR blocks of one VPC to avoid overlap.

• Adjusted the network routing tables to ensure traffic was correctly routed.

- Lessons Learned:

Ensure that CIDR blocks are carefully planned to avoid conflicts in hybrid cloud environments.

- How to Avoid:

• Plan CIDR blocks in advance to ensure they do not overlap.

• Review and validate network configurations during the planning phase of hybrid cloud setups.

Scenario #179: Service Discovery Failures Due to Network Policy Blocking DNS Traffic

- Category: Networking

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: Service discovery failed when a network policy was mistakenly applied to block DNS traffic, preventing pods from resolving services within the cluster.

- What Happened:A network policy was applied to restrict traffic between namespaces but unintentionally blocked DNS traffic on UDP port 53, causing service discovery to fail.

- Diagnosis Steps:

• Ran kubectl get networkpolicy and found an ingress rule that blocked UDP traffic.

• Used kubectl exec to test DNS resolution inside the affected pods, which confirmed that DNS queries were being blocked.

Root Cause: The network policy unintentionally blocked DNS traffic due to a misconfigured ingress rule.

- Fix/Workaround:

• Updated the network policy to allow DNS traffic on UDP port 53.

• Restarted the affected pods to restore service discovery functionality.

- Lessons Learned:

Always carefully test network policies to ensure they don't inadvertently block critical traffic like DNS.

- How to Avoid:

• Review and test network policies thoroughly before applying them in production.

• Implement automated tests to verify that critical services like DNS are not affected by policy changes.

Scenario #180: Intermittent Network Connectivity Due to Overloaded Overlay Network

- Category: Networking

- Environment: K8s v1.19, OpenStack

Scenario

- Summary: Pods experienced intermittent network connectivity issues due to an overloaded overlay network that could not handle the traffic.

- What Happened:An overlay network (Flannel) used to connect pods was overwhelmed due to high traffic volume, resulting in intermittent packet drops and network congestion.

- Diagnosis Steps:

• Used kubectl exec to trace packet loss between pods and detected intermittent connectivity.

• Monitored network interfaces and observed high traffic volume and congestion on the overlay network.

Root Cause: The overlay network (Flannel) could not handle the traffic load due to insufficient resources allocated to the network component.

- Fix/Workaround:

• Reconfigured the overlay network to use a more scalable network plugin.

• Increased resource allocation for the network components and scaled the infrastructure to handle the load.

- Lessons Learned:

Ensure that network plugins are properly configured and scaled to handle the expected traffic volume.

- How to Avoid:

• Monitor network traffic patterns and adjust resource allocation as needed.

• Consider using more scalable network plugins for high-traffic workloads.

Scenario #181: Pod-to-Pod Communication Failure Due to CNI Plugin Configuration Issue

- Category: Networking

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Pods were unable to communicate with each other due to a misconfiguration in the CNI plugin.

- What Happened:The Calico CNI plugin configuration was missing the necessary IP pool definitions, which caused pods to fail to obtain IPs from the defined pool, resulting in communication failure between pods.

- Diagnosis Steps:

  • Ran kubectl describe pod to identify that the pods had no assigned IP addresses.

  • Inspected the CNI plugin logs and identified missing IP pool configurations.

  Root Cause: The IP pool was not defined in the Calico CNI plugin configuration, causing pods to be unable to get network addresses.

- Fix/Workaround:

  • Updated the Calico configuration to include the correct IP pool definitions.

  • Restarted the affected pods to obtain new IPs.

- Lessons Learned:

  Always verify CNI plugin configuration, especially IP pool settings, before deploying a cluster.

- How to Avoid:

  • Automate the verification of CNI configurations during cluster setup.

  • Test network functionality before scaling applications.

Scenario #182: Sporadic DNS Failures Due to Resource Contention in CoreDNS Pods

- Category: Networking

- Environment: K8s v1.19, GKE

Scenario

- Summary: Sporadic DNS resolution failures occurred due to resource contention in CoreDNS pods, which were not allocated enough CPU resources.

- What Happened:CoreDNS pods were experiencing sporadic failures due to high CPU utilization. DNS resolution intermittently failed during peak load times.

- Diagnosis Steps:

• Used kubectl top pod to monitor resource usage and found that CoreDNS pods were CPU-bound.

• Monitored DNS query logs and found a correlation between high CPU usage and DNS resolution failures.

Root Cause: CoreDNS pods were not allocated sufficient CPU resources to handle the DNS query load during peak times.

- Fix/Workaround:

• Increased CPU resource requests and limits for CoreDNS pods.

• Enabled horizontal pod autoscaling for CoreDNS to scale during high demand.

- Lessons Learned:

CoreDNS should be adequately resourced, and autoscaling should be enabled to handle varying DNS query loads.

- How to Avoid:

• Set proper resource requests and limits for CoreDNS.

• Implement autoscaling for DNS services based on real-time load.

Scenario #183: High Latency in Pod-to-Node Communication Due to Overlay Network

- Category: Networking

- Environment: K8s v1.21, OpenShift

  Scenario

- Summary: High latency was observed in pod-to-node communication due to network overhead introduced by the overlay network.

- What Happened:The cluster was using Flannel as the CNI plugin, and network latency increased as the overlay network was unable to efficiently handle the traffic between pods and nodes.

- Diagnosis Steps:

  • Used kubectl exec to measure network latency between pods and nodes.

  • Analyzed the network traffic and identified high latency due to the overlay network's encapsulation.

  Root Cause: The Flannel overlay network introduced additional overhead, which caused latency in pod-to-node communication.

- Fix/Workaround:

  • Switched to a different CNI plugin (Calico) that offered better performance for the network topology.

• Retested pod-to-node communication after switching CNI plugins.

- Lessons Learned:

Choose the right CNI plugin based on network performance needs, especially in high-throughput environments.

- How to Avoid:

• Perform a performance evaluation of different CNI plugins during cluster planning.

• Monitor network performance regularly and switch plugins if necessary.

Scenario #184: Service Discovery Issues Due to DNS Cache Staleness

- Category: Networking

- Environment: K8s v1.20, On-premise

Scenario

- Summary: Service discovery failed due to stale DNS cache entries that were not updated when services changed IPs.

- What Happened:The DNS resolver cached the old IP addresses for services, causing service discovery failures when the IPs of the services changed.

- Diagnosis Steps:

• Used kubectl exec to verify DNS cache entries.

• Observed that the cached IPs were outdated and did not reflect the current service IPs.

Root Cause: The DNS cache was not being properly refreshed, causing stale DNS entries.

- Fix/Workaround:

• Cleared the DNS cache manually and implemented shorter TTL (Time-To-Live) values for DNS records.

• Restarted CoreDNS pods to apply changes.

- Lessons Learned:

Ensure that DNS TTL values are appropriately set to avoid stale cache issues.

- How to Avoid:

• Regularly monitor DNS cache and refresh TTL values to ensure up-to-date resolution.

• Implement a caching strategy that works well with Kubernetes service discovery.

Scenario #185: Network Partition Between Node Pools in Multi-Zone Cluster

- Category: Networking

- Environment: K8s v1.18, GKE

Scenario

- Summary: Pods in different node pools located in different zones experienced network partitioning due to a misconfigured regional load balancer.

- What Happened:The regional load balancer was not properly configured to handle traffic between node pools located in different zones, causing network partitioning between pods in different zones.

- Diagnosis Steps:

• Used kubectl exec to verify pod-to-pod communication between node pools and found packet loss.

• Inspected the load balancer configuration and found that cross-zone traffic was not properly routed.

Root Cause: The regional load balancer was misconfigured, blocking traffic between nodes in different zones.

- Fix/Workaround:

• Updated the regional load balancer configuration to properly route cross-zone traffic.

• Re-deployed the affected pods to restore connectivity.

- Lessons Learned:

Ensure proper configuration of load balancers to support multi-zone communication in cloud environments.

- How to Avoid:

• Test multi-zone communication setups thoroughly before going into production.

• Automate the validation of load balancer configurations.

Scenario #186: Pod Network Isolation Failure Due to Missing NetworkPolicy

- Category: Networking

- Environment: K8s v1.21, AKS

  Scenario

- Summary: Pods that were intended to be isolated from each other could communicate freely due to a missing NetworkPolicy.

- What Happened:The project had requirements for strict pod isolation, but the necessary NetworkPolicy was not created, resulting in unexpected communication between pods that should not have had network access to each other.

- Diagnosis Steps:

  • Inspected kubectl get networkpolicy and found no policies defined for pod isolation.

  • Verified pod-to-pod communication and observed that pods in different namespaces could communicate without restriction.

  Root Cause: Absence of a NetworkPolicy meant that all pods had default access to one another.

- Fix/Workaround:

  • Created appropriate NetworkPolicy to restrict pod communication based on the namespace and labels.

  • Applied the NetworkPolicy and tested communication to ensure isolation was working.

- Lessons Learned:

Always implement and test network policies when security and isolation are a concern.

- How to Avoid:

• Implement strict NetworkPolicy from the outset when dealing with sensitive workloads.

• Automate the validation of network policies during CI/CD pipeline deployment.

Scenario #187: Flapping Node Network Connectivity Due to MTU Mismatch

- Category: Networking

- Environment: K8s v1.20, On-Premise

Scenario

- Summary: Nodes in the cluster were flapping due to mismatched MTU settings between Kubernetes and the underlying physical network, causing intermittent network connectivity issues.

- What Happened:The physical network's MTU was configured differently from the MTU settings in the Kubernetes CNI plugin, causing packet fragmentation. As a result, node-to-node communication was sporadic.

- Diagnosis Steps:

• Used kubectl describe node and checked the node’s network configuration.

• Verified the MTU settings in the physical network and compared them to the Kubernetes settings, which were mismatched.

Root Cause: The mismatch in MTU settings caused fragmentation, resulting in unreliable connectivity between nodes.

- Fix/Workaround:

• Updated the Kubernetes network plugin's MTU setting to match the physical network MTU.

• Restarted the affected nodes and validated the network stability.

- Lessons Learned:

Ensure that the MTU setting in the CNI plugin matches the physical network's MTU to avoid connectivity issues.

- How to Avoid:

• Always verify the MTU settings in both the physical network and the CNI plugin during cluster setup.

• Include network performance testing in your cluster validation procedures.

Scenario #188: DNS Query Timeout Due to Unoptimized CoreDNS Config

- Category: Networking

- Environment: K8s v1.18, GKE

Scenario

- Summary: DNS queries were timing out in the cluster, causing delays in service discovery, due to unoptimized CoreDNS configuration.

- What Happened:The CoreDNS configuration was not optimized for the cluster size, resulting in DNS query timeouts under high load.

- Diagnosis Steps:

• Checked CoreDNS logs and saw frequent query timeouts.

• Used kubectl describe pod on CoreDNS pods and found that they were under-resourced, leading to DNS query delays.

Root Cause: CoreDNS was misconfigured and lacked adequate CPU and memory resources to handle the query load.

- Fix/Workaround:

• Increased CPU and memory requests/limits for CoreDNS.

• Optimized the CoreDNS configuration to use a more efficient query handling strategy.

- Lessons Learned:

CoreDNS needs to be properly resourced and optimized for performance, especially in large clusters.

- How to Avoid:

• Regularly monitor DNS performance and adjust CoreDNS resource allocations.

• Fine-tune the CoreDNS configuration to improve query handling efficiency.

Scenario #189: Traffic Splitting Failure Due to Incorrect Service LoadBalancer Configuration

- Category: Networking

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Traffic splitting between two microservices failed due to a misconfiguration in the Service LoadBalancer.

- What Happened:The load balancing rules were incorrectly set up for the service, which caused requests to only route to one instance of a microservice, despite the intention to split traffic between two.

- Diagnosis Steps:

• Used kubectl describe svc to inspect the Service configuration and discovered incorrect annotations for traffic splitting.

• Analyzed AWS load balancer logs and saw that traffic was directed to only one pod.

Root Cause: Misconfigured traffic splitting annotations in the Service definition prevented the load balancer from distributing traffic correctly.

- Fix/Workaround:

• Corrected the annotations in the Service definition to enable proper traffic splitting.

• Redeployed the Service and tested that traffic was split as expected.

- Lessons Learned:

  Always double-check load balancer and service annotations when implementing traffic splitting in a microservices environment.

- How to Avoid:

  • Test traffic splitting configurations in a staging environment before applying them in production.

  • Automate the verification of load balancer and service configurations.

Scenario #190: Network Latency Between Pods in Different Regions

- Category: Networking

- Environment: K8s v1.19, Azure AKS

  Scenario

- Summary: Pods in different Azure regions experienced high network latency, affecting application performance.

- What Happened:The Kubernetes cluster spanned multiple Azure regions, but the inter-region networking was not optimized, resulting in significant network latency between pods in different regions.

- Diagnosis Steps:

• Used kubectl exec to measure ping times between pods in different regions and observed high latency.

• Inspected Azure network settings and found that there were no specific optimizations in place for inter-region traffic.

Root Cause: Lack of inter-region network optimization and reliance on default settings led to high latency between regions.

- Fix/Workaround:

• Configured Azure Virtual Network peering with appropriate bandwidth settings.

• Enabled specific network optimizations for inter-region communication.

- Lessons Learned:

When deploying clusters across multiple regions, network latency should be carefully managed and optimized.

- How to Avoid:

• Use region-specific optimizations and peering when deploying multi-region clusters.

• Test the network performance before and after cross-region deployments to ensure acceptable latency.

Scenario #191: Port Collision Between Services Due to Missing Port Ranges

- Category: Networking

- Environment: K8s v1.21, AKS

Scenario

- Summary: Two services attempted to bind to the same port, causing a port collision and service failures.

- What Happened:The services were configured without specifying unique port ranges, and both attempted to use the same port on the same node, leading to port binding issues.

- Diagnosis Steps:

• Used kubectl get svc to check the services' port configurations and found that both services were trying to bind to the same port.

• Verified node logs and observed port binding errors.

Root Cause: Missing port range configurations in the service definitions led to port collision.

- Fix/Workaround:

• Updated the service definitions to specify unique ports or port ranges.

• Redeployed the services to resolve the conflict.

- Lessons Learned:

Always ensure that services use unique port configurations to avoid conflicts.

- How to Avoid:

• Define port ranges explicitly in service configurations.

• Use tools like kubectl to validate port allocations before deploying services.

Scenario #192: Pod-to-External Service Connectivity Failures Due to Egress Network Policy

- Category: Networking

- Environment: K8s v1.20, AWS EKS

Scenario

- Summary: Pods failed to connect to an external service due to an overly restrictive egress network policy.

- What Happened:An egress network policy was too restrictive and blocked traffic from the pods to external services, leading to connectivity issues.

- Diagnosis Steps:

• Used kubectl describe networkpolicy to inspect egress rules and found that the policy was blocking all outbound traffic.

• Verified connectivity to the external service and confirmed the network policy was the cause.

Root Cause: An overly restrictive egress network policy prevented pods from accessing external services.

- Fix/Workaround:

• Modified the egress network policy to allow traffic to the required external service.

• Applied the updated policy and tested connectivity.

- Lessons Learned:

Be mindful when applying network policies, especially egress rules that affect external connectivity.

- How to Avoid:

• Test network policies in a staging environment before applying them in production.

• Implement gradual rollouts for network policies to avoid wide-scale disruptions.

Scenario #193: Pod Connectivity Loss After Network Plugin Upgrade

- Category: Networking

- Environment: K8s v1.18, GKE

Scenario

- Summary: Pods lost connectivity after an upgrade of the Calico network plugin due to misconfigured IP pool settings.

- What Happened:After upgrading the Calico CNI plugin, the IP pool configuration was not correctly migrated, which caused pods to lose connectivity to other pods and services.

- Diagnosis Steps:

• Checked kubectl describe pod and found that the pods were not assigned IPs.

• Inspected Calico configuration and discovered that the IP pool settings were not properly carried over during the upgrade.

Root Cause: The upgrade process failed to migrate the IP pool configuration, leading to network connectivity issues for the pods.

- Fix/Workaround:

• Manually updated the Calico configuration to restore the correct IP pool settings.

• Restarted the Calico pods and verified pod connectivity.

- Lessons Learned:

Ensure network plugin upgrades are carefully tested and configurations are validated after upgrades.

- How to Avoid:

• Perform network plugin upgrades in a staging environment before applying to production.

• Use configuration management tools to keep track of network plugin settings.

Scenario #194: External DNS Not Resolving After Cluster Network Changes

- Category: Networking

- Environment: K8s v1.19, DigitalOcean

Scenario

- Summary: External DNS resolution stopped working after changes were made to the cluster network configuration.

- What Happened:After modifying the CNI configuration and reconfiguring IP ranges, external DNS resolution failed for services outside the cluster.

- Diagnosis Steps:

  • Checked DNS resolution inside the cluster using kubectl exec and found that internal DNS queries were working, but external queries were failing.

  • Verified DNS resolver configuration and noticed that the external DNS forwarders were misconfigured after network changes.

  Root Cause: The external DNS forwarder settings were not correctly updated after network changes.

- Fix/Workaround:

  • Updated CoreDNS configuration to correctly forward DNS queries to external DNS servers.

  • Restarted CoreDNS pods to apply changes.

- Lessons Learned:

  Network configuration changes can impact DNS settings, and these should be verified post-change.

- How to Avoid:

  • Implement automated DNS validation tests to ensure external DNS resolution works after network changes.

  • Document and verify DNS configurations before and after network changes.

Scenario #195: Slow Pod Communication Due to Misconfigured MTU in Network Plugin

- Category: Networking

- Environment: K8s v1.22, On-premise

  Scenario

- Summary: Pod-to-pod communication was slow due to an incorrect MTU setting in the network plugin.

- What Happened:The network plugin was configured with an MTU that did not match the underlying network's MTU, leading to packet fragmentation and slower communication between pods.

- Diagnosis Steps:

  • Used ping to check latency between pods and observed unusually high latency.

  • Inspected the network plugin’s MTU configuration and compared it with the host’s MTU, discovering a mismatch.

  Root Cause: The MTU setting in the network plugin was too high, causing packet fragmentation and slow communication.

- Fix/Workaround:

  • Corrected the MTU setting in the network plugin to match the host’s MTU.

  • Restarted the affected pods to apply the changes.

- Lessons Learned:

Ensure that MTU settings are aligned between the network plugin and the underlying network infrastructure.

- How to Avoid:

• Review and validate MTU settings when configuring network plugins.

• Use monitoring tools to detect network performance issues like fragmentation.

Scenario #196: High CPU Usage in Nodes Due to Overloaded Network Plugin

- Category: Networking

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Nodes experienced high CPU usage due to an overloaded network plugin that couldn’t handle traffic spikes effectively.

- What Happened:The network plugin was designed to handle a certain volume of traffic, but when the pod-to-pod communication increased, the plugin was unable to scale efficiently, leading to high CPU consumption.

- Diagnosis Steps:

• Monitored node metrics with kubectl top nodes and noticed unusually high CPU usage on affected nodes.

• Checked logs for the network plugin and found evidence of resource exhaustion under high traffic conditions.

Root Cause: The network plugin was not adequately resourced to handle high traffic spikes, leading to resource exhaustion.

- Fix/Workaround:

• Increased resource allocation (CPU/memory) for the network plugin.

• Configured scaling policies for the network plugin to dynamically adjust resources.

- Lessons Learned:

Network plugins need to be able to scale in response to increased traffic to prevent performance degradation.

- How to Avoid:

• Regularly monitor network plugin performance and resources.

• Configure auto-scaling and adjust resource allocation based on traffic patterns.

Scenario #197: Cross-Namespace Network Isolation Not Enforced

- Category: Networking

- Environment: K8s v1.19, OpenShift

Scenario

- Summary: Network isolation between namespaces failed due to an incorrectly applied NetworkPolicy.

- What Happened:The NetworkPolicy intended to isolate communication between namespaces was not enforced because it was misconfigured.

- Diagnosis Steps:

  • Checked the NetworkPolicy with kubectl describe networkpolicy and found that the selector was too broad, allowing communication across namespaces.

  • Verified namespace communication and found that pods in different namespaces could still communicate freely.

  Root Cause: The NetworkPolicy selectors were too broad, and isolation was not enforced between namespaces.

- Fix/Workaround:

  • Refined the NetworkPolicy to more specifically target pods within certain namespaces.

  • Re-applied the updated NetworkPolicy and validated the isolation.

- Lessons Learned:

  Ensure that NetworkPolicy selectors are specific to prevent unintended communication.

- How to Avoid:

  • Always validate network policies before deploying to production.

  • Use namespace-specific selectors to enforce isolation when necessary.

Scenario #198: Inconsistent Service Discovery Due to CoreDNS Misconfiguration

- Category: Networking

- Environment: K8s v1.20, GKE

Scenario

- Summary: Service discovery was inconsistent due to misconfigured CoreDNS settings.

- What Happened:The CoreDNS configuration was updated to use an external resolver, but the external resolver had intermittent issues, leading to service discovery failures.

- Diagnosis Steps:

• Checked CoreDNS logs with kubectl logs -n kube-system <coredns-pod> and noticed errors with the external resolver.

• Used kubectl get svc to check service names and found that some services could not be resolved reliably.

Root Cause: Misconfigured external DNS resolver in CoreDNS caused service discovery failures.

- Fix/Workaround:

• Reverted CoreDNS configuration to use the internal DNS resolver instead of the external one.

• Restarted CoreDNS pods to apply the changes.

- Lessons Learned:

  External DNS resolvers can introduce reliability issues; test these changes carefully.

- How to Avoid:

  • Use internal DNS resolvers for core service discovery within the cluster.

  • Implement monitoring for DNS resolution health.

Scenario #199: Network Segmentation Issues Due to Misconfigured CNI

- Category: Networking

- Environment: K8s v1.18, IBM Cloud

  Scenario

- Summary: Network segmentation between clusters failed due to incorrect CNI (Container Network Interface) plugin configuration.

- What Happened:The CNI plugin was incorrectly configured, allowing pods from different network segments to communicate, violating security requirements.

- Diagnosis Steps:

  • Inspected kubectl describe node and found that nodes were assigned to multiple network segments.

• Used network monitoring tools to verify that pods in different segments were able to communicate.

Root Cause: The CNI plugin was not correctly segmented between networks, allowing unauthorized communication.

- Fix/Workaround:

• Reconfigured the CNI plugin to enforce correct network segmentation.

• Applied the changes and tested communication between pods from different segments.

- Lessons Learned:

Network segmentation configurations should be thoroughly reviewed to prevent unauthorized communication.

- How to Avoid:

• Implement strong isolation policies in the network plugin.

• Regularly audit network configurations and validate segmentation between clusters.

Scenario #200: DNS Cache Poisoning in CoreDNS

- Category: Networking

- Environment: K8s v1.23, DigitalOcean

Scenario

- Summary: DNS cache poisoning occurred in CoreDNS, leading to incorrect IP resolution for services.

- What Happened:A malicious actor compromised a DNS record by injecting a false IP address into the CoreDNS cache, causing services to resolve to an incorrect IP.

- Diagnosis Steps:

  • Monitored CoreDNS logs and identified suspicious query patterns.

  • Used kubectl exec to inspect the DNS cache and found that some services had incorrect IP addresses cached.

  Root Cause: CoreDNS cache was not sufficiently secured, allowing for DNS cache poisoning.

- Fix/Workaround:

  • Implemented DNS query validation and hardened CoreDNS security by limiting cache lifetime and introducing DNSSEC.

  • Cleared the DNS cache and restarted CoreDNS to remove the poisoned entries.

- Lessons Learned:

  Securing DNS caching is critical to prevent cache poisoning attacks.

- How to Avoid:

  • Use DNSSEC or other DNS security mechanisms to validate responses.

  • Regularly monitor and audit CoreDNS logs for anomalies.

3. Security

Scenario #201: Unauthorized Access to Secrets Due to Incorrect RBAC Permissions

- Category: Security

- Environment: K8s v1.22, GKE

Scenario

- Summary: Unauthorized users were able to access Kubernetes secrets due to overly permissive RBAC roles.

- What Happened:A service account was granted cluster-admin permissions, which allowed users to access sensitive secrets via kubectl. This led to a security breach when one of the users exploited the permissions.

- Diagnosis Steps:

  • Inspected RBAC roles with kubectl get roles and kubectl get clusterroles to identify misconfigured roles.

  • Checked logs and found that sensitive secrets were accessed using a service account that shouldn't have had access.

Root Cause: The service account was granted excessive permissions via RBAC roles.

- Fix/Workaround:

  • Reconfigured RBAC roles to adhere to the principle of least privilege.

  • Limited the permissions of the service account and tested access controls.

- Lessons Learned:

Always follow the principle of least privilege when configuring RBAC for service accounts and users.

- How to Avoid:

  • Regularly audit RBAC roles and service account permissions.

  • Implement role-based access control (RBAC) with tight restrictions on who can access secrets.

Scenario #202: Insecure Network Policies Leading to Pod Exposure

- Category: Security

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: Pods intended to be isolated were exposed to unauthorized traffic due to misconfigured network policies.

- What Happened:A network policy was meant to block communication between pods in different namespaces, but it was misconfigured, allowing unauthorized access between pods.

- Diagnosis Steps:

• Used kubectl get networkpolicy to check existing network policies.

• Observed that the network policy's podSelector was incorrectly configured, allowing access between pods from different namespaces.

Root Cause: Misconfigured NetworkPolicy selectors allowed unwanted access between pods.

- Fix/Workaround:

• Corrected the NetworkPolicy by refining podSelector and applying stricter isolation.

• Tested the updated policy to confirm proper isolation between namespaces.

- Lessons Learned:

Network policies must be carefully crafted to prevent unauthorized access between pods.

- How to Avoid:

• Implement and test network policies in a staging environment before applying to production.

• Regularly audit network policies to ensure they align with security requirements.

Scenario #203: Privileged Container Vulnerability Due to Incorrect Security Context

- Category: Security

- Environment: K8s v1.21, Azure AKS

Scenario

- Summary: A container running with elevated privileges due to an incorrect security context exposed the cluster to potential privilege escalation attacks.

- What Happened:A container was configured with privileged: true in its security context, which allowed it to gain elevated permissions and access sensitive parts of the node.

- Diagnosis Steps:

  • Inspected the pod security context with kubectl describe pod and found that the container was running as a privileged container.

  • Cross-referenced the container's security settings with the deployment YAML and identified the privileged: true setting.

  Root Cause: Misconfigured security context allowed the container to run with elevated privileges, leading to security risks.

- Fix/Workaround:

  • Removed privileged: true from the container's security context.

  • Applied the updated deployment and monitored the pod for any security incidents.

- Lessons Learned:

  Always avoid using privileged: true unless absolutely necessary for certain workloads.

- How to Avoid:

  • Review security contexts in deployment configurations to ensure containers are not running with excessive privileges.

  • Implement automated checks to flag insecure container configurations.

Scenario #204: Exposed Kubernetes Dashboard Due to Misconfigured Ingress

- Category: Security

- Environment: K8s v1.20, GKE

  Scenario

- Summary: The Kubernetes dashboard was exposed to the public internet due to a misconfigured Ingress resource.

- What Happened:The Ingress resource for the Kubernetes dashboard was incorrectly set up to allow external traffic from all IPs, making the dashboard accessible without authentication.

- Diagnosis Steps:

  • Used kubectl describe ingress to inspect the Ingress resource configuration.

  • Found that the Ingress had no restrictions on IP addresses, allowing anyone with the URL to access the dashboard.

  Root Cause: Misconfigured Ingress resource with open access to the Kubernetes dashboard.

- Fix/Workaround:

  • Updated the Ingress resource to restrict access to specific IP addresses or require authentication for access.

  • Re-applied the updated configuration and tested access controls.

- Lessons Learned:

  Always secure the Kubernetes dashboard by restricting access to trusted IPs or requiring strong authentication.

- How to Avoid:

  • Apply strict network policies or use ingress controllers with authentication for access to the Kubernetes dashboard.

  • Regularly review Ingress resources for security misconfigurations.

Scenario #205: Unencrypted Communication Between Pods Due to Missing TLS Configuration

- Category: Security

- Environment: K8s v1.18, On-Premise

  Scenario

- Summary: Communication between microservices in the cluster was not encrypted due to missing TLS configuration, exposing data to potential interception.

- What Happened:The microservices were communicating over HTTP instead of HTTPS, and there was no mutual TLS (mTLS) configured for secure communication, making data vulnerable to interception.

- Diagnosis Steps:

• Reviewed service-to-service communication with network monitoring tools and found that HTTP was being used instead of HTTPS.

• Inspected the Ingress and service definitions and found that no TLS secrets or certificates were configured.

Root Cause: Lack of TLS configuration for service communication led to unencrypted communication.

- Fix/Workaround:

• Configured mTLS between services to ensure encrypted communication.

• Deployed certificates and updated services to use HTTPS for communication.

- Lessons Learned:

Secure communication between microservices is crucial to prevent data leakage or interception.

- How to Avoid:

• Always configure TLS for service-to-service communication, especially for sensitive workloads.

• Automate the generation and renewal of certificates.

Scenario #206: Sensitive Data in Logs Due to Improper Log Sanitization

- Category: Security

- Environment: K8s v1.23, Azure AKS

Scenario

- Summary: Sensitive data, such as API keys and passwords, was logged due to improper sanitization in application logs.

- What Happened:A vulnerability in the application caused API keys and secrets to be included in logs, which were not sanitized before being stored in the central logging system.

- Diagnosis Steps:

• Examined the application logs using kubectl logs and found that sensitive data was included in plain text.

• Inspected the logging configuration and found that there were no filters in place to scrub sensitive data.

Root Cause: Lack of proper sanitization in the logging process allowed sensitive data to be exposed.

- Fix/Workaround:

• Updated the application to sanitize sensitive data before it was logged.

• Configured the logging system to filter out sensitive information from logs.

- Lessons Learned:

Sensitive data should never be included in logs in an unencrypted or unsanitized format.

- How to Avoid:

• Implement log sanitization techniques to ensure that sensitive information is never exposed in logs.

• Regularly audit logging configurations to ensure that they are secure.

Scenario #207: Insufficient Pod Security Policies Leading to Privilege Escalation

- Category: Security

- Environment: K8s v1.21, GKE

Scenario

- Summary: Privilege escalation was possible due to insufficiently restrictive PodSecurityPolicies (PSPs).

- What Happened:The PodSecurityPolicy (PSP) was not configured to prevent privilege escalation, allowing containers to run with excessive privileges and exploit vulnerabilities within the cluster.

- Diagnosis Steps:

• Inspected the PSPs using kubectl get psp and noticed that the allowPrivilegeEscalation flag was set to true.

• Cross-referenced the pod configurations and found that containers were running with root privileges and escalated privileges.

Root Cause: Insufficiently restrictive PodSecurityPolicies allowed privilege escalation.

- Fix/Workaround:

• Updated the PSPs to restrict privilege escalation by setting allowPrivilegeEscalation: false.

• Applied the updated policies and tested pod deployments to confirm proper restrictions.

- Lessons Learned:

Always configure restrictive PodSecurityPolicies to prevent privilege escalation within containers.

- How to Avoid:

• Regularly review and apply restrictive PSPs to enforce security best practices in the cluster.

• Use automated tools to enforce security policies on all pods and containers.

Scenario #208: Service Account Token Compromise

- Category: Security

- Environment: K8s v1.22, DigitalOcean

Scenario

- Summary: A compromised service account token was used to gain unauthorized access to the cluster's API server.

- What Happened:A service account token was leaked through an insecure deployment configuration, allowing attackers to gain unauthorized access to the Kubernetes API server.

- Diagnosis Steps:

• Analyzed the audit logs and identified that the compromised service account token was being used to make API calls.

• Inspected the deployment YAML and found that the service account token was exposed as an environment variable.

Root Cause: Exposing the service account token in environment variables allowed it to be compromised.

- Fix/Workaround:

• Rotated the service account token and updated the deployment to prevent exposure.

• Used Kubernetes secrets management to securely store sensitive tokens.

- Lessons Learned:

Never expose sensitive tokens or secrets through environment variables or unsecured channels.

- How to Avoid:

• Use Kubernetes Secrets to store sensitive information securely.

• Regularly rotate service account tokens and audit access logs for suspicious activity.

Scenario #209: Lack of Regular Vulnerability Scanning in Container Images

- Category: Security

- Environment: K8s v1.19, On-Premise

Scenario

- Summary: The container images used in the cluster were not regularly scanned for vulnerabilities, leading to deployment of vulnerable images.

- What Happened:A critical vulnerability in one of the base images was discovered after deployment, as no vulnerability scanning tools were used to validate the images before use.

- Diagnosis Steps:

• Checked the container image build pipeline and confirmed that no vulnerability scanning tools were integrated.

• Analyzed the CVE database and identified that a vulnerability in the image was already known.

Root Cause: Lack of regular vulnerability scanning in the container image pipeline.

- Fix/Workaround:

• Integrated a vulnerability scanning tool like Clair or Trivy into the CI/CD pipeline.

• Rebuilt the container images with a fixed version and redeployed them.

- Lessons Learned:

Regular vulnerability scanning of container images is essential to ensure secure deployments.

- How to Avoid:

• Integrate automated vulnerability scanning tools into the container build process.

• Perform regular image audits and keep base images updated.

Scenario #210: Insufficient Container Image Signing Leading to Unverified Deployments

- Category: Security

- Environment: K8s v1.20, Google Cloud

Scenario

- Summary: Unverified container images were deployed due to the lack of image signing, exposing the cluster to potential malicious code.

- What Happened:Malicious code was deployed when a container image was pulled from a public registry without being properly signed or verified.

- Diagnosis Steps:

• Checked the image pull policies and found that image signing was not enabled for the container registry.

• Inspected the container image and found that it had not been signed.

Root Cause: Lack of image signing led to the deployment of unverified images.

- Fix/Workaround:

• Enabled image signing in the container registry and integrated it with Kubernetes for secure image verification.

• Re-pulled and deployed only signed images to the cluster.

- Lessons Learned:

Always use signed images to ensure the integrity and authenticity of containers being deployed.

- How to Avoid:

• Implement image signing as part of the container build and deployment pipeline.

• Regularly audit deployed container images to verify their integrity.

Scenario #211: Insecure Default Namespace Leading to Unauthorized Access

- Category: Security

- Environment: K8s v1.22, AWS EKS

Scenario

- Summary: Unauthorized users gained access to resources in the default namespace due to lack of namespace isolation.

- What Happened:Users without explicit permissions accessed and modified resources in the default namespace because the default namespace was not protected by network policies or RBAC rules.

- Diagnosis Steps:

  • Checked RBAC policies and confirmed that users had access to resources in the default namespace.

  • Inspected network policies and found no restrictions on traffic to/from the default namespace.

  Root Cause: Insufficient access control to the default namespace allowed unauthorized access.

- Fix/Workaround:

  • Restricted access to the default namespace using RBAC and network policies.

  • Created separate namespaces for different workloads and applied appropriate isolation policies.

- Lessons Learned:

  Avoid using the default namespace for critical resources and ensure that proper access control and isolation are in place.

- How to Avoid:

  • Use dedicated namespaces for different workloads with appropriate RBAC and network policies.

  • Regularly audit namespace access and policies.

Scenario #212: Vulnerable OpenSSL Version in Container Images

- Category: Security

- Environment: K8s v1.21, DigitalOcean

Scenario

- Summary: A container image was using an outdated and vulnerable version of OpenSSL, exposing the cluster to known security vulnerabilities.

- What Happened:A critical vulnerability in OpenSSL was discovered after deploying a container that had not been updated to use a secure version of the library.

- Diagnosis Steps:

• Analyzed the Dockerfile and confirmed the container image was based on an outdated version of OpenSSL.

• Cross-referenced the CVE database and identified that the version used in the container had known vulnerabilities.

Root Cause: The container image was built with an outdated version of OpenSSL that contained unpatched vulnerabilities.

- Fix/Workaround:

• Rebuilt the container image using a newer, secure version of OpenSSL.

• Deployed the updated image and monitored for any further issues.

- Lessons Learned:

Always ensure that containers are built using updated and patched versions of libraries to mitigate known vulnerabilities.

- How to Avoid:

• Integrate automated vulnerability scanning tools into the CI/CD pipeline to identify outdated or vulnerable dependencies.

• Regularly update container base images to the latest secure versions.

Scenario #213: Misconfigured API Server Authentication Allowing External Access

- Category: Security

- Environment: K8s v1.20, GKE

Scenario

- Summary: API server authentication was misconfigured, allowing external unauthenticated users to access the Kubernetes API.

- What Happened:The Kubernetes API server was mistakenly exposed without authentication, allowing external users to query resources without any credentials.

- Diagnosis Steps:

• Examined the API server configuration and found that the authentication was set to allow unauthenticated access (--insecure-allow-any-token was enabled).

• Reviewed ingress controllers and firewall rules and confirmed that the API server was publicly accessible.

Root Cause: The API server was misconfigured to allow unauthenticated access, exposing the cluster to unauthorized requests.

- Fix/Workaround:

• Disabled unauthenticated access by removing --insecure-allow-any-token from the API server configuration.

• Configured proper authentication methods, such as client certificates or OAuth2.

- Lessons Learned:

Always secure the Kubernetes API server and ensure proper authentication is in place to prevent unauthorized access.

- How to Avoid:

• Regularly audit the API server configuration to ensure proper authentication mechanisms are enabled.

• Use firewalls and access controls to limit access to the API server.

Scenario #214: Insufficient Node Security Due to Lack of OS Hardening

- Category: Security

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: Nodes in the cluster were insecure due to a lack of proper OS hardening, making them vulnerable to attacks.

- What Happened:The nodes in the cluster were not properly hardened according to security best practices, leaving them vulnerable to potential exploitation.

- Diagnosis Steps:

• Conducted a security audit of the nodes and identified unpatched vulnerabilities in the operating system.

• Verified that security settings like SSH root login and password authentication were not properly disabled.

Root Cause: Insufficient OS hardening on the nodes exposed them to security risks.

- Fix/Workaround:

• Applied OS hardening guidelines, such as disabling root SSH access and ensuring only key-based authentication.

• Updated the operating system with the latest security patches.

- Lessons Learned:

Proper OS hardening is essential for securing Kubernetes nodes and reducing the attack surface.

- How to Avoid:

• Implement automated checks to enforce OS hardening settings across all nodes.

• Regularly update nodes with the latest security patches.

Scenario #215: Unrestricted Ingress Access to Sensitive Resources

- Category: Security

- Environment: K8s v1.21, GKE

Scenario

- Summary: Sensitive services were exposed to the public internet due to unrestricted ingress rules.

- What Happened:An ingress resource was misconfigured, exposing sensitive internal services such as the Kubernetes dashboard and internal APIs to the public.

- Diagnosis Steps:

• Inspected the ingress rules and found that they allowed traffic from all IPs (host: *).

• Confirmed that the services were critical and should not have been exposed to external traffic.

Root Cause: Misconfigured ingress resource allowed unrestricted access to sensitive services.

- Fix/Workaround:

• Restrict ingress traffic by specifying allowed IP ranges or adding authentication for access to sensitive resources.

• Used a more restrictive ingress controller and verified that access was limited to trusted sources.

- Lessons Learned:

Always secure ingress access to critical resources by applying proper access controls.

- How to Avoid:

• Regularly review and audit ingress configurations to prevent exposing sensitive services.

• Implement access control lists (ACLs) and authentication for sensitive endpoints.

Scenario #216: Exposure of Sensitive Data in Container Environment Variables

- Category: Security

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: Sensitive data, such as database credentials, was exposed through environment variables in container configurations.

- What Happened:Sensitive environment variables containing credentials were directly included in Kubernetes deployment YAML files, making them visible to anyone with access to the deployment.

- Diagnosis Steps:

• Examined the deployment manifests and discovered sensitive data in the environment variables section.

• Used kubectl describe deployment and found that credentials were stored in plain text in the environment section of containers.

Root Cause: Storing sensitive data in plaintext environment variables exposed it to unauthorized users.

- Fix/Workaround:

• Moved sensitive data into Kubernetes Secrets instead of directly embedding them in environment variables.

• Updated the deployment YAML to reference the Secrets and applied the changes.

- Lessons Learned:

Sensitive data should always be stored securely in Kubernetes Secrets or external secret management systems.

- How to Avoid:

• Use Kubernetes Secrets for storing sensitive data like passwords, API keys, and certificates.

• Regularly audit configurations to ensure secrets are not exposed in plain text.

Scenario #217: Inadequate Container Resource Limits Leading to DoS Attacks

- Category: Security

- Environment: K8s v1.20, On-Premise

Scenario

- Summary: A lack of resource limits on containers allowed a denial-of-service (DoS) attack to disrupt services by consuming excessive CPU and memory.

- What Happened:A container without resource limits was able to consume all available CPU and memory on the node, causing other containers to become unresponsive and leading to a denial-of-service (DoS).

- Diagnosis Steps:

• Monitored resource usage with kubectl top pods and identified a container consuming excessive resources.

• Inspected the deployment and found that resource limits were not set for the container.

Root Cause: Containers without resource limits allowed resource exhaustion, which led to a DoS situation.

- Fix/Workaround:

• Set appropriate resource requests and limits in the container specification to prevent resource exhaustion.

• Applied resource quotas to limit the total resource usage for namespaces.

- Lessons Learned:

Always define resource requests and limits to ensure containers do not overconsume resources and cause instability.

- How to Avoid:

• Apply resource requests and limits to all containers.

• Monitor resource usage and set appropriate quotas to prevent resource abuse.

Scenario #218: Exposure of Container Logs Due to Insufficient Log Management

- Category: Security

- Environment: K8s v1.21, Google Cloud

Scenario

- Summary: Container logs were exposed to unauthorized users due to insufficient log management controls.

- What Happened:Logs were stored in plain text and exposed to users who should not have had access, revealing sensitive data like error messages and stack traces.

- Diagnosis Steps:

  • Reviewed log access permissions and found that they were too permissive, allowing unauthorized users to access logs.

  • Checked the log storage system and found logs were being stored unencrypted.

Root Cause: Insufficient log management controls led to unauthorized access to sensitive logs.

- Fix/Workaround:

  • Implemented access controls to restrict log access to authorized users only.

  • Encrypted logs at rest and in transit to prevent exposure.

- Lessons Learned:

Logs should be securely stored and access should be restricted to authorized personnel only.

- How to Avoid:

• Implement access control and encryption for logs.

• Regularly review log access policies to ensure security best practices are followed.

Scenario #219: Using Insecure Docker Registry for Container Images

- Category: Security

- Environment: K8s v1.18, On-Premise

Scenario

- Summary: The cluster was pulling container images from an insecure, untrusted Docker registry, exposing the system to the risk of malicious images.

- What Happened:The Kubernetes cluster was configured to pull images from an untrusted Docker registry, which lacked proper security measures such as image signing or vulnerability scanning.

- Diagnosis Steps:

• Inspected the image pull configuration and found that the registry URL pointed to an insecure registry.

• Analyzed the images and found they lacked proper security scans or signing.

Root Cause: Using an insecure registry without proper image signing and scanning introduced the risk of malicious images.

- Fix/Workaround:

• Configured Kubernetes to pull images only from trusted and secure registries.

• Implemented image signing and vulnerability scanning in the CI/CD pipeline.

- Lessons Learned:

Always use trusted and secure Docker registries and implement image security practices.

- How to Avoid:

• Use secure image registries with image signing and vulnerability scanning enabled.

• Implement image whitelisting to control where container images can be pulled from.

Scenario #220: Weak Pod Security Policies Leading to Privileged Containers

- Category: Security

- Environment: K8s v1.19, AWS EKS

Scenario

- Summary: Privileged containers were deployed due to weak or missing Pod Security Policies (PSPs), exposing the cluster to security risks.

- What Happened:The absence of strict Pod Security Policies allowed containers to run with elevated privileges, leading to a potential security risk as malicious pods could gain unauthorized access to node resources.

- Diagnosis Steps:

 • Inspected the cluster configuration and found that PSPs were either missing or improperly configured.

 • Verified that certain containers were running as privileged, which allowed them to access kernel-level resources.

 Root Cause: Weak or missing Pod Security Policies allowed privileged containers to be deployed without restriction.

- Fix/Workaround:

 • Created and applied strict Pod Security Policies to limit the permissions of containers.

 • Enforced the use of non-privileged containers for sensitive workloads.

- Lessons Learned:

 Strict Pod Security Policies are essential for securing containers and limiting the attack surface.

- How to Avoid:

 • Implement and enforce strong Pod Security Policies to limit the privileges of containers.

 • Regularly audit containers to ensure they do not run with unnecessary privileges.

Scenario #221: Unsecured Kubernetes Dashboard

- Category: Security

- Environment: K8s v1.21, GKE

Scenario

- Summary: The Kubernetes Dashboard was exposed to the public internet without proper authentication or access controls, allowing unauthorized users to access sensitive cluster information.

- What Happened:The Kubernetes Dashboard was deployed without proper access control or authentication mechanisms, leaving it open to the internet and allowing unauthorized users to access sensitive cluster data.

- Diagnosis Steps:

• Checked the Dashboard configuration and found that the kubectl proxy option was used without authentication enabled.

• Verified that the Dashboard was accessible via the internet without any IP restrictions.

Root Cause: The Kubernetes Dashboard was exposed without proper authentication or network restrictions.

- Fix/Workaround:

• Enabled authentication and RBAC rules for the Kubernetes Dashboard.

• Restricted access to the Dashboard by allowing connections only from trusted IP addresses.

- Lessons Learned:

Always secure the Kubernetes Dashboard with authentication and limit access using network policies.

- How to Avoid:

  • Configure proper authentication for the Kubernetes Dashboard.

  • Use network policies to restrict access to sensitive resources like the Dashboard.

Scenario #222: Using HTTP Instead of HTTPS for Ingress Resources

- Category: Security

- Environment: K8s v1.22, Google Cloud

  Scenario

- Summary: Sensitive applications were exposed using HTTP instead of HTTPS, leaving communication vulnerable to eavesdropping and man-in-the-middle attacks.

- What Happened:Sensitive application traffic was served over HTTP rather than HTTPS, allowing attackers to potentially intercept or manipulate traffic.

- Diagnosis Steps:

  • Inspected ingress resource configurations and confirmed that TLS termination was not configured.

  • Verified that sensitive endpoints were exposed over HTTP without encryption.

  Root Cause: Lack of TLS encryption in the ingress resources exposed sensitive traffic to security risks.

- Fix/Workaround:

• Configured ingress controllers to use HTTPS by setting up TLS termination with valid SSL certificates.

• Redirected all HTTP traffic to HTTPS to ensure encrypted communication.

- Lessons Learned:

Always use HTTPS for secure communication between clients and Kubernetes applications, especially for sensitive data.

- How to Avoid:

• Configure TLS termination for all ingress resources to encrypt traffic.

• Regularly audit ingress resources to ensure that sensitive applications are protected by HTTPS.

Scenario #223: Insecure Network Policies Exposing Internal Services

- Category: Security

- Environment: K8s v1.20, On-Premise

Scenario

- Summary: Network policies were too permissive, exposing internal services to unnecessary access, increasing the risk of lateral movement within the cluster.

- What Happened:Network policies were overly permissive, allowing services within the cluster to communicate with each other without restriction. This made it easier for attackers to move laterally if they compromised one service.

- Diagnosis Steps:

• Reviewed the network policy configurations and found that most services were allowed to communicate with any other service within the cluster.

• Inspected the logs for unauthorized connections between services.

Root Cause: Permissive network policies allowed unnecessary communication between services, increasing the potential attack surface.

- Fix/Workaround:

• Restricted network policies to only allow communication between services that needed to interact.

• Used namespace-based segmentation and ingress/egress rules to enforce tighter security.

- Lessons Learned:

Proper network segmentation and restrictive network policies are crucial for securing the internal traffic between services.

- How to Avoid:

• Apply the principle of least privilege when defining network policies, ensuring only necessary communication is allowed.

• Regularly audit network policies to ensure they are as restrictive as needed.

Scenario #224: Exposing Sensitive Secrets in Environment Variables

- Category: Security

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Sensitive credentials were stored in environment variables within the pod specification, exposing them to potential attackers.

- What Happened:Sensitive data such as database passwords and API keys were stored as environment variables in plain text within Kubernetes pod specifications, making them accessible to anyone who had access to the pod's configuration.

- Diagnosis Steps:

  • Examined the pod specification files and found that sensitive credentials were stored as environment variables in plaintext.

  • Verified that no secrets management solution like Kubernetes Secrets was being used to handle sensitive data.

  Root Cause: Sensitive data was stored insecurely in environment variables rather than using Kubernetes Secrets or an external secrets management solution.

- Fix/Workaround:

  • Moved sensitive data to Kubernetes Secrets and updated the pod configurations to reference the secrets.

  • Ensured that secrets were encrypted and only accessible by the relevant services.

- Lessons Learned:

  Always store sensitive data securely using Kubernetes Secrets or an external secrets management solution, and avoid embedding it in plain text.

- How to Avoid:

• Use Kubernetes Secrets to store sensitive data and reference them in your deployments.

• Regularly audit your configuration files to ensure sensitive data is not exposed in plaintext.

Scenario #225: Insufficient RBAC Permissions Leading to Unauthorized Access

- Category: Security

- Environment: K8s v1.20, On-Premise

Scenario

- Summary: Insufficient Role-Based Access Control (RBAC) configurations allowed unauthorized users to access and modify sensitive resources within the cluster.

- What Happened:The RBAC configurations were not properly set up, granting more permissions than necessary. As a result, unauthorized users were able to access sensitive resources such as secrets, config maps, and deployments.

- Diagnosis Steps:

• Reviewed RBAC policies and roles and found that users had been granted broad permissions, including access to sensitive namespaces and resources.

• Verified that the principle of least privilege was not followed.

Root Cause: RBAC roles were not properly configured, resulting in excessive permissions being granted to users.

- Fix/Workaround:

• Reconfigured RBAC roles to ensure that users only had the minimum necessary permissions.

• Applied the principle of least privilege and limited access to sensitive resources.

- Lessons Learned:

RBAC should be configured according to the principle of least privilege to minimize security risks.

- How to Avoid:

• Regularly review and audit RBAC configurations to ensure they align with the principle of least privilege.

• Implement strict role definitions and limit access to only the resources necessary for each user.

Scenario #226: Insecure Ingress Controller Exposed to the Internet

- Category: Security

- Environment: K8s v1.22, Google Cloud

Scenario

- Summary: An insecure ingress controller was exposed to the internet, allowing attackers to exploit vulnerabilities in the controller.

- What Happened:An ingress controller was deployed with insufficient security hardening and exposed to the public internet, making it a target for potential exploits.

- Diagnosis Steps:

  • Examined the ingress controller configuration and found that it was publicly exposed without adequate access controls.

  • Identified that no authentication or IP whitelisting was in place to protect the ingress controller.

  Root Cause: Insufficient security configurations on the ingress controller allowed it to be exposed to the internet.

- Fix/Workaround:

  • Secured the ingress controller by implementing proper authentication and IP whitelisting.

  • Ensured that only authorized users or services could access the ingress controller.

- Lessons Learned:

  Always secure ingress controllers with authentication and limit access using network policies or IP whitelisting.

- How to Avoid:

  • Configure authentication for ingress controllers and restrict access to trusted IPs.

  • Regularly audit ingress configurations to ensure they are secure.

Scenario #227: Lack of Security Updates in Container Images

- Category: Security

- Environment: K8s v1.19, DigitalOcean

  Scenario

- Summary: The cluster was running outdated container images without the latest security patches, exposing it to known vulnerabilities.

- What Happened:The container images used in the cluster had not been updated with the latest security patches, making them vulnerable to known exploits.

- Diagnosis Steps:

  • Analyzed the container images and found that they had not been updated in months.

  • Checked for known vulnerabilities in the base image and discovered unpatched CVEs.

  Root Cause: Container images were not regularly updated with the latest security patches.

- Fix/Workaround:

  • Rebuilt the container images with updated base images and security patches.

  • Implemented a policy for regularly updating container images to include the latest security fixes.

- Lessons Learned:

Regular updates to container images are essential for maintaining security and reducing the risk of vulnerabilities.

- How to Avoid:

  • Implement automated image scanning and patching as part of the CI/CD pipeline.

  • Regularly review and update container images to ensure they include the latest security patches.

Scenario #228: Exposed Kubelet API Without Authentication

- Category: Security

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: The Kubelet API was exposed without proper authentication or authorization, allowing external users to query cluster node details.

- What Happened:The Kubelet API was inadvertently exposed to the internet without authentication, making it possible for unauthorized users to access sensitive node information, such as pod logs and node status.

- Diagnosis Steps:

  • Checked Kubelet API configurations and confirmed that no authentication mechanisms (e.g., client certificates) were in place.

• Verified that Kubelet was exposed via a public-facing load balancer without any IP whitelisting.

Root Cause: Lack of authentication and network restrictions for the Kubelet API exposed it to unauthorized access.

- Fix/Workaround:

• Restricted Kubelet API access to internal networks by updating security group rules.

• Enabled authentication and authorization for the Kubelet API using client certificates.

- Lessons Learned:

Always secure the Kubelet API with authentication and restrict access to trusted IPs or internal networks.

- How to Avoid:

• Use network policies to block access to the Kubelet API from the public internet.

• Enforce authentication on the Kubelet API using client certificates or other mechanisms.

Scenario #229: Inadequate Logging of Sensitive Events

- Category: Security

- Environment: K8s v1.22, Google Cloud

Scenario

- Summary: Sensitive security events were not logged, preventing detection of potential security breaches or misconfigurations.

- What Happened:Security-related events, such as privilege escalations and unauthorized access attempts, were not being logged correctly due to misconfigurations in the auditing system.

- Diagnosis Steps:

 • Examined the audit policy configuration and found that critical security events (e.g., access to secrets, changes in RBAC) were not being captured.

 • Reviewed Kubernetes logs and discovered the absence of certain expected security events.

 Root Cause: Misconfigured Kubernetes auditing policies prevented sensitive security events from being logged.

- Fix/Workaround:

 • Reconfigured the Kubernetes audit policy to capture sensitive events, including user access to secrets, privilege escalations, and changes in RBAC roles.

 • Integrated log aggregation and alerting tools to monitor security logs in real time.

- Lessons Learned:

 Properly configuring audit logging is essential for detecting potential security incidents and ensuring compliance.

- How to Avoid:

 • Implement comprehensive audit logging policies to capture sensitive security events.

 • Regularly review audit logs and integrate with centralized monitoring solutions for real-time alerts.

Scenario #230: Misconfigured RBAC Allowing Cluster Admin Privileges to Developers

- Category: Security

- Environment: K8s v1.19, On-Premise

  Scenario

- Summary: Developers were mistakenly granted cluster admin privileges due to misconfigured RBAC roles, which gave them the ability to modify sensitive resources.

- What Happened:The RBAC configuration allowed developers to assume roles with cluster admin privileges, enabling them to access and modify sensitive resources, including secrets and critical configurations.

- Diagnosis Steps:

  • Reviewed RBAC roles and bindings and found that developers had been granted roles with broader privileges than required.

  • Examined audit logs to confirm that developers had accessed resources outside of their designated scope.

  Root Cause: Misconfigured RBAC roles allowed developers to acquire cluster admin privileges, leading to unnecessary access to sensitive resources.

- Fix/Workaround:

  • Reconfigured RBAC roles to follow the principle of least privilege and removed cluster admin permissions for developers.

• Implemented role separation to ensure developers only had access to resources necessary for their tasks.

- Lessons Learned:

  Always follow the principle of least privilege when assigning roles, and regularly audit RBAC configurations to prevent privilege escalation.

- How to Avoid:

  • Regularly review and audit RBAC configurations to ensure that only the minimum necessary permissions are granted to each user.

  • Use namespaces and role-based access controls to enforce separation of duties and limit access to sensitive resources.

Scenario #231: Insufficiently Secured Service Account Permissions

- Category: Security

- Environment: K8s v1.20, AWS EKS

  Scenario

- Summary: Service accounts were granted excessive permissions, giving pods access to resources they did not require, leading to a potential security risk.

- What Happened:A service account used by multiple pods had broader permissions than needed. This allowed one compromised pod to access sensitive resources across the cluster, including secrets and privileged services.

- Diagnosis Steps:

 • Audited service account configurations and found that many pods were using the same service account with excessive permissions.

 • Investigated the logs and identified that the compromised pod was able to access restricted resources.

 Root Cause: Service accounts were granted overly broad permissions, violating the principle of least privilege.

- Fix/Workaround:

 • Created specific service accounts for each pod with minimal necessary permissions.

 • Applied strict RBAC rules to restrict access to sensitive resources for service accounts.

- Lessons Learned:

 Use fine-grained permissions for service accounts to reduce the impact of a compromise.

- How to Avoid:

 • Regularly audit service accounts and ensure they follow the principle of least privilege.

 • Implement namespace-level access control to limit service account scope.

Scenario #232: Cluster Secrets Exposed Due to Insecure Mounting

- Category: Security

- Environment: K8s v1.21, On-Premise

Scenario

- Summary: Kubernetes secrets were mounted into pods insecurely, exposing sensitive information to unauthorized users.

- What Happened:Secrets were mounted directly into the filesystem of pods, making them accessible to anyone with access to the pod's filesystem, including attackers who compromised the pod.

- Diagnosis Steps:

• Inspected pod configurations and found that secrets were mounted in plain text into the pod’s filesystem.

• Verified that no access control policies were in place for secret access.

Root Cause: Secrets were mounted without sufficient access control, allowing them to be exposed in the pod filesystem.

- Fix/Workaround:

• Moved secrets to Kubernetes Secrets and mounted them using environment variables instead of directly into the filesystem.

• Restricted access to secrets using RBAC and implemented encryption for sensitive data.

- Lessons Learned:

Always use Kubernetes Secrets for sensitive information and ensure proper access control.

- How to Avoid:

• Mount secrets as environment variables rather than directly into the filesystem.

• Use encryption and access controls to limit exposure of sensitive data.

Scenario #233: Improperly Configured API Server Authorization

- Category: Security

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: The Kubernetes API server was improperly configured, allowing unauthorized users to make API calls without proper authorization.

- What Happened:The API server authorization mechanisms were misconfigured, allowing unauthorized users to bypass RBAC rules and access sensitive cluster resources.

- Diagnosis Steps:

• Reviewed the API server configuration and found that the authorization mode was incorrectly set, allowing certain users to bypass RBAC.

• Verified access control logs and confirmed unauthorized actions.

Root Cause: Misconfiguration in the API server’s authorization mode allowed unauthorized API calls.

- Fix/Workaround:

• Reconfigured the API server to use proper authorization mechanisms (e.g., RBAC, ABAC).

• Validated and tested API server access to ensure only authorized users could make API calls.

- Lessons Learned:

Properly configuring the Kubernetes API server's authorization mechanism is crucial for cluster security.

- How to Avoid:

• Regularly audit API server configurations, especially authorization modes, to ensure proper access control.

• Implement strict RBAC and ABAC policies for fine-grained access control.

Scenario #234: Compromised Image Registry Access Credentials

- Category: Security

- Environment: K8s v1.19, On-Premise

Scenario

- Summary: The image registry access credentials were compromised, allowing attackers to pull and run malicious images in the cluster.

- What Happened:The credentials used to access the container image registry were stored in plaintext in a config map, and these credentials were stolen by an attacker, who then pulled a malicious container image into the cluster.

- Diagnosis Steps:

• Reviewed configuration files and discovered the registry access credentials were stored in plaintext within a config map.

• Analyzed logs and found that a malicious image had been pulled from the compromised registry.

Root Cause: Storing sensitive credentials in plaintext made them vulnerable to theft and misuse.

- Fix/Workaround:

• Moved credentials to Kubernetes Secrets, which are encrypted by default.

• Enforced the use of trusted image registries and scanned images for vulnerabilities before use.

- Lessons Learned:

Sensitive credentials should never be stored in plaintext; Kubernetes Secrets provide secure storage.

- How to Avoid:

• Always use Kubernetes Secrets to store sensitive information like image registry credentials.

• Implement image scanning and whitelisting policies to ensure only trusted images are deployed.

Scenario #235: Insufficiently Secured Cluster API Server Access

- Category: Security

- Environment: K8s v1.23, Google Cloud

Scenario

- Summary: The API server was exposed with insufficient security, allowing unauthorized external access and increasing the risk of exploitation.

- What Happened:The Kubernetes API server was configured to allow access from external IP addresses without proper security measures such as encryption or authentication, which could be exploited by attackers.

- Diagnosis Steps:

• Inspected the API server's ingress configuration and found it was not restricted to internal networks or protected by encryption.

• Checked for authentication mechanisms and found that none were properly enforced for external requests.

Root Cause: Inadequate protection of the Kubernetes API server allowed unauthenticated external access.

- Fix/Workaround:

• Restrict access to the API server using firewall rules to allow only internal IP addresses.

• Implemented TLS encryption and client certificate authentication for secure access.

- Lessons Learned:

Always secure the Kubernetes API server with proper network restrictions, encryption, and authentication.

- How to Avoid:

• Use firewall rules and IP whitelisting to restrict access to the API server.

• Enforce encryption and authentication for all external access to the API server.

Scenario #236: Misconfigured Admission Controllers Allowing Insecure Resources

- Category: Security

- Environment: K8s v1.21, AWS EKS

Scenario

- Summary: Admission controllers were misconfigured, allowing the creation of insecure or non-compliant resources.

- What Happened:Admission controllers were either not enabled or misconfigured, allowing users to create resources without enforcing security standards, such as running containers with privileged access or without required security policies.

- Diagnosis Steps:

• Reviewed the admission controller configuration and found that key controllers like PodSecurityPolicy and LimitRanger were either disabled or misconfigured.

• Audited resources and found that insecure pods were being created without restrictions.

Root Cause: Misconfigured or missing admission controllers allowed insecure resources to be deployed.

- Fix/Workaround:

• Enabled and properly configured necessary admission controllers, such as PodSecurityPolicy and LimitRanger, to enforce security policies during resource creation.

• Regularly audited resource creation and applied security policies to avoid insecure configurations.

- Lessons Learned:

Admission controllers are essential for enforcing security standards and preventing insecure resources from being created.

- How to Avoid:

• Ensure that key admission controllers are enabled and configured correctly.

• Regularly audit the use of admission controllers and enforce best practices for security policies.

Scenario #237: Lack of Security Auditing and Monitoring in Cluster

- Category: Security

- Environment: K8s v1.22, DigitalOcean

Scenario

- Summary: The lack of proper auditing and monitoring allowed security events to go undetected, resulting in delayed response to potential security threats.

- What Happened:The cluster lacked a comprehensive auditing and monitoring solution, and there were no alerts configured for sensitive security events, such as privilege escalations or suspicious activities.

- Diagnosis Steps:

  • Checked the audit logging configuration and found that it was either incomplete or disabled.

  • Verified that no centralized logging or monitoring solutions were in place for security events.

Root Cause: Absence of audit logging and real-time monitoring prevented timely detection of potential security issues.

- Fix/Workaround:

  • Implemented audit logging and integrated a centralized logging and monitoring solution, such as Prometheus and ELK stack, to detect security incidents.

  • Set up alerts for suspicious activities and security violations.

- Lessons Learned:

Continuous monitoring and auditing are essential for detecting and responding to security incidents.

- How to Avoid:

• Enable and configure audit logging to capture security-related events.

• Set up real-time monitoring and alerting for security threats.

Scenario #238: Exposed Internal Services Due to Misconfigured Load Balancer

- Category: Security

- Environment: K8s v1.19, On-Premise

Scenario

- Summary: Internal services were inadvertently exposed to the public due to incorrect load balancer configurations, leading to potential security risks.

- What Happened:A load balancer was misconfigured, exposing internal services to the public internet without proper access controls, increasing the risk of unauthorized access.

- Diagnosis Steps:

• Reviewed the load balancer configuration and found that internal services were exposed to external traffic.

• Identified that no authentication or access control was in place for the exposed services.

Root Cause: Incorrect load balancer configuration exposed internal services to the internet.

- Fix/Workaround:

• Reconfigured the load balancer to restrict access to internal services, ensuring that only authorized users or services could connect.

• Implemented authentication and IP whitelisting to secure the exposed services.

- Lessons Learned:

Always secure internal services exposed via load balancers by applying strict access controls and authentication.

- How to Avoid:

• Review and verify load balancer configurations regularly to ensure no unintended exposure.

• Implement network policies and access controls to secure internal services.

Scenario #239: Kubernetes Secrets Accessed via Insecure Network

- Category: Security

- Environment: K8s v1.20, GKE

Scenario

- Summary: Kubernetes secrets were accessed via an insecure network connection, exposing sensitive information to unauthorized parties.

- What Happened:Secrets were transmitted over an unsecured network connection between pods and the Kubernetes API server, allowing an attacker to intercept the data.

- Diagnosis Steps:

• Inspected network traffic and found that Kubernetes API server connections were not encrypted (HTTP instead of HTTPS).

• Analyzed pod configurations and found that sensitive secrets were being transmitted without encryption.

Root Cause: Lack of encryption for sensitive data in transit allowed it to be intercepted.

- Fix/Workaround:

• Configured Kubernetes to use HTTPS for all API server communications.

• Ensured that all pod-to-API server traffic was encrypted and used secure protocols.

- Lessons Learned:

Always encrypt traffic between Kubernetes components, especially when transmitting sensitive data like secrets.

- How to Avoid:

• Ensure HTTPS is enforced for all communications between Kubernetes components.

• Use Transport Layer Security (TLS) for secure communication across the cluster.

Scenario #240: Pod Security Policies Not Enforced

- Category: Security

- Environment: K8s v1.21, On-Premise

Scenario

- Summary: Pod security policies were not enforced, allowing the deployment of pods with unsafe configurations, such as privileged access and host network use.

- What Happened:The PodSecurityPolicy (PSP) feature was disabled or misconfigured, allowing pods with privileged access to be deployed. This opened up the cluster to potential privilege escalation and security vulnerabilities.

- Diagnosis Steps:

• Inspected the PodSecurityPolicy settings and found that no PSPs were defined or enabled.

• Checked recent deployments and found pods with host network access and privileged containers.

Root Cause: Disabled or misconfigured PodSecurityPolicy allowed unsafe pods to be deployed.

- Fix/Workaround:

• Enabled and configured PodSecurityPolicy to enforce security controls, such as preventing privileged containers or host network usage.

• Audited existing pod configurations and updated them to comply with security policies.

- Lessons Learned:

Enforcing PodSecurityPolicies is crucial for securing pod configurations and preventing risky deployments.

- How to Avoid:

• Enable and properly configure PodSecurityPolicy to restrict unsafe pod configurations.

• Regularly audit pod configurations to ensure compliance with security standards.

Scenario #241: Unpatched Vulnerabilities in Cluster Nodes

- Category: Security

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: Cluster nodes were not regularly patched, exposing known vulnerabilities that were later exploited by attackers.

- What Happened:The Kubernetes cluster nodes were running outdated operating system versions with unpatched security vulnerabilities. These vulnerabilities were exploited in a targeted attack, compromising the nodes and enabling unauthorized access.

- Diagnosis Steps:

• Conducted a security audit of the nodes and identified several unpatched operating system vulnerabilities.

• Reviewed cluster logs and found evidence of unauthorized access attempts targeting known vulnerabilities.

Root Cause: Lack of regular patching of cluster nodes allowed known vulnerabilities to be exploited.

- Fix/Workaround:

• Patches were applied to all affected nodes to fix known vulnerabilities.

• Established a regular patch management process to ensure that cluster nodes were kept up to date.

- Lessons Learned:

Regular patching of Kubernetes nodes and underlying operating systems is essential for preventing security exploits.

- How to Avoid:

• Implement automated patching and vulnerability scanning for cluster nodes.

• Regularly review security advisories and apply patches promptly.

Scenario #242: Weak Network Policies Allowing Unrestricted Traffic

- Category: Security

- Environment: K8s v1.18, On-Premise

Scenario

- Summary: Network policies were not properly configured, allowing unrestricted traffic between pods, which led to lateral movement by attackers after a pod was compromised.

- What Happened:Insufficient network policies were in place, allowing all pods to communicate freely with each other. This enabled attackers who compromised one pod to move laterally across the cluster and access additional services.

- Diagnosis Steps:

  • Reviewed existing network policies and found that none were in place or were too permissive.

  • Conducted a security assessment and identified pods with excessive permissions to communicate with critical services.

  Root Cause: Lack of restrictive network policies allowed unrestricted traffic between pods, increasing the attack surface.

- Fix/Workaround:

  • Created strict network policies to control pod-to-pod communication, limiting access to sensitive services.

  • Regularly reviewed and updated network policies to minimize exposure.

- Lessons Learned:

  Proper network segmentation with Kubernetes network policies is essential to prevent lateral movement in case of a breach.

- How to Avoid:

  • Implement network policies that restrict communication between pods, especially for sensitive services.

  • Regularly audit and update network policies to ensure they align with security best practices.

Scenario #243: Exposed Dashboard Without Authentication

- Category: Security

- Environment: K8s v1.19, GKE

  Scenario

- Summary: Kubernetes dashboard was exposed to the internet without authentication, allowing unauthorized users to access cluster information and potentially take control.

- What Happened:The Kubernetes Dashboard was exposed to the public internet without proper authentication or authorization mechanisms, allowing attackers to view sensitive cluster information and even execute actions like deploying malicious workloads.

- Diagnosis Steps:

  • Verified that the Kubernetes Dashboard was exposed via an insecure ingress.

  • Discovered that no authentication or role-based access controls (RBAC) were applied to restrict access.

  Root Cause: Misconfiguration of the Kubernetes Dashboard exposure settings allowed it to be publicly accessible.

- Fix/Workaround:

  • Restricted access to the Kubernetes Dashboard by securing the ingress and requiring authentication via RBAC or OAuth.

  • Implemented a VPN and IP whitelisting to ensure that only authorized users could access the dashboard.

- Lessons Learned:

Always secure the Kubernetes Dashboard with proper authentication mechanisms and limit exposure to trusted users.

- How to Avoid:

• Use authentication and authorization to protect access to the Kubernetes Dashboard.

• Apply proper ingress and network policies to prevent exposure of critical services.

Scenario #244: Use of Insecure Container Images

- Category: Security

- Environment: K8s v1.20, AWS EKS

Scenario

- Summary: Insecure container images were used in production, leading to the deployment of containers with known vulnerabilities.

- What Happened:Containers were pulled from an untrusted registry that did not implement image scanning. These images had known security vulnerabilities, which were exploited once deployed in the cluster.

- Diagnosis Steps:

• Reviewed container image sourcing and found that some images were pulled from unverified registries.

• Scanned the images for vulnerabilities and identified several critical issues, including outdated libraries and unpatched vulnerabilities.

Root Cause: Use of untrusted and insecure container images led to the deployment of containers with vulnerabilities.

- Fix/Workaround:

• Enforced the use of trusted container image registries that support vulnerability scanning.

• Integrated image scanning tools like Trivy or Clair into the CI/CD pipeline to identify vulnerabilities before deployment.

- Lessons Learned:

Always verify and scan container images for vulnerabilities before using them in production.

- How to Avoid:

• Use trusted image registries and always scan container images for vulnerabilities before deploying them.

• Implement an image signing and verification process to ensure image integrity.

Scenario #245: Misconfigured TLS Certificates

- Category: Security

- Environment: K8s v1.23, Azure AKS

Scenario

- Summary: Misconfigured TLS certificates led to insecure communication between Kubernetes components, exposing the cluster to potential attacks.

- What Happened:TLS certificates used for internal communication between Kubernetes components were either expired or misconfigured, leading to insecure communication channels.

- Diagnosis Steps:

• Inspected TLS certificate expiration dates and found that many certificates had expired or were incorrectly configured.

• Verified logs and found that some internal communication channels were using unencrypted HTTP due to certificate issues.

Root Cause: Expired or misconfigured TLS certificates allowed unencrypted communication between Kubernetes components.

- Fix/Workaround:

• Regenerated and replaced expired certificates.

• Configured Kubernetes components to use valid TLS certificates for all internal communications.

- Lessons Learned:

Regularly monitor and rotate TLS certificates to ensure secure communication within the cluster.

- How to Avoid:

• Set up certificate expiration monitoring and automate certificate renewal.

• Regularly audit and update the Kubernetes cluster's TLS certificates.

Scenario #246: Excessive Privileges for Service Accounts

- Category: Security

- Environment: K8s v1.22, Google Cloud

Scenario

- Summary: Service accounts were granted excessive privileges, allowing them to perform operations outside their intended scope, increasing the risk of compromise.

- What Happened:Service accounts were assigned broad permissions that allowed them to perform sensitive actions, such as modifying cluster configurations and accessing secret resources.

- Diagnosis Steps:

• Audited RBAC configurations and identified several service accounts with excessive privileges.

• Cross-referenced service account usage with pod deployment and confirmed unnecessary access.

Root Cause: Overly permissive RBAC roles and service account configurations granted excessive privileges.

- Fix/Workaround:

• Updated RBAC roles to follow the principle of least privilege, ensuring service accounts only had the minimum necessary permissions.

• Regularly audited service accounts to verify proper access control.

- Lessons Learned:

Service accounts should follow the principle of least privilege to limit the impact of any compromise.

- How to Avoid:

• Review and restrict service account permissions regularly to ensure they have only the necessary privileges.

• Implement role-based access control (RBAC) policies that enforce strict access control.

Scenario #247: Exposure of Sensitive Logs Due to Misconfigured Logging Setup

- Category: Security

- Environment: K8s v1.21, DigitalOcean

Scenario

- Summary: Sensitive logs, such as those containing authentication tokens and private keys, were exposed due to a misconfigured logging setup.

- What Happened:The logging setup was not configured to redact sensitive data, and logs containing authentication tokens and private keys were accessible to unauthorized users.

- Diagnosis Steps:

  • Inspected log configurations and found that logs were being stored without redaction or filtering of sensitive data.

  • Verified that sensitive log data was accessible through centralized logging systems.

  Root Cause: Misconfigured logging setup allowed sensitive data to be stored and viewed without proper redaction.

- Fix/Workaround:

  • Updated log configuration to redact or filter sensitive data, such as tokens and private keys, before storing logs.

  • Implemented access controls to restrict who can view logs and what data is exposed.

- Lessons Learned:

  Always ensure that sensitive data in logs is either redacted or filtered to prevent unintentional exposure.

- How to Avoid:

  • Configure logging systems to automatically redact sensitive data before storing it.

  • Apply access controls to logging systems to limit access to sensitive log data.

Scenario #248: Use of Deprecated APIs with Known Vulnerabilities

- Category: Security

- Environment: K8s v1.19, AWS EKS

 Scenario

- Summary: The cluster was using deprecated Kubernetes APIs that contained known security vulnerabilities, which were exploited by attackers.

- What Happened:Kubernetes components and applications in the cluster were using deprecated APIs, which were no longer supported and contained known security issues. The attacker exploited these vulnerabilities to gain unauthorized access to sensitive resources.

- Diagnosis Steps:

 • Reviewed the API versions used by the cluster components and identified deprecated APIs.

 • Scanned cluster logs and found unauthorized access attempts tied to these deprecated API calls.

 Root Cause: Outdated and deprecated APIs were used, exposing the cluster to security vulnerabilities that were no longer patched.

- Fix/Workaround:

 • Upgraded Kubernetes components and applications to use supported and secure API versions.

 • Removed deprecated API usage and enforced only supported versions.

- Lessons Learned:

Always stay current with supported APIs and avoid using deprecated versions that may not receive security patches.

- How to Avoid:

  • Regularly check Kubernetes API deprecation notices and migrate to supported API versions.
  • Set up monitoring to detect the use of deprecated APIs in your cluster.

Scenario #249: Lack of Security Context in Pod Specifications

- Category: Security

- Environment: K8s v1.22, Google Cloud

  Scenario

- Summary: Pods were deployed without defining appropriate security contexts, resulting in privileged containers and access to host resources.

- What Happened:Many pods in the cluster were deployed without specifying a security context, leading to some containers running with excessive privileges, such as access to the host network or running as root. This allowed attackers to escalate privileges if they were able to compromise a container.

- Diagnosis Steps:

  • Inspected pod specifications and identified a lack of security context definitions, allowing containers to run as root or with other high privileges.

• Verified pod logs and found containers with host network access and root user privileges.

Root Cause: Failure to specify a security context for pods allowed containers to run with unsafe permissions.

- Fix/Workaround:

• Defined and enforced security contexts for all pod deployments to restrict privilege escalation and limit access to sensitive resources.

• Implemented security policies to reject pods that do not comply with security context guidelines.

- Lessons Learned:

Always define security contexts for pods to enforce proper security boundaries.

- How to Avoid:

• Set default security contexts for all pod deployments.

• Use Kubernetes admission controllers to ensure that only secure pod configurations are allowed.

Scenario #250: Compromised Container Runtime

- Category: Security

- Environment: K8s v1.21, On-Premise

Scenario

- Summary: The container runtime (Docker) was compromised, allowing an attacker to gain control over the containers running on the node.

- What Happened:A vulnerability in the container runtime was exploited by an attacker, who was able to execute arbitrary code on the host node. This allowed the attacker to escape the container and execute malicious commands on the underlying infrastructure.

- Diagnosis Steps:

  • Detected unusual activity on the node using intrusion detection systems (IDS).

  • Analyzed container runtime logs and discovered signs of container runtime compromise.

  • Found that the attacker exploited a known vulnerability in the Docker daemon to gain elevated privileges.

  Root Cause: An unpatched vulnerability in the container runtime allowed an attacker to escape the container and gain access to the host.

- Fix/Workaround:

  • Immediately patched the container runtime (Docker) to address the security vulnerability.

  • Implemented security measures, such as running containers with user namespaces and seccomp profiles to minimize the impact of any future exploits.

- Lessons Learned:

  Regularly update the container runtime and other components to mitigate the risk of known vulnerabilities.

- How to Avoid:

• Keep the container runtime up to date with security patches.

• Use security features like seccomp, AppArmor, or SELinux to minimize container privileges and limit potential attack vectors.

Scenario #251: Insufficient RBAC Permissions for Cluster Admin

- Category: Security

- Environment: K8s v1.22, GKE

Scenario

- Summary: A cluster administrator was mistakenly granted insufficient RBAC permissions, preventing them from performing essential management tasks.

- What Happened:A new RBAC policy was applied, which inadvertently restricted the cluster admin's ability to manage critical components such as deployments, services, and namespaces. This caused operational issues and hindered the ability to scale or fix issues in the cluster.

- Diagnosis Steps:

• Audited the RBAC policy and identified restrictive permissions applied to the admin role.

• Attempted various management tasks and encountered "forbidden" errors when accessing critical cluster resources.

Root Cause: Misconfiguration in the RBAC policy prevented the cluster admin from accessing necessary resources.

- Fix/Workaround:

  • Updated the RBAC policy to ensure that the cluster admin role had the correct permissions to manage all resources.

  • Implemented a more granular RBAC policy review process to avoid future issues.

- Lessons Learned:

  Always test RBAC configurations in a staging environment to avoid accidental misconfigurations.

- How to Avoid:

  • Implement automated RBAC policy checks and enforce least privilege principles.

  • Regularly review and update RBAC roles to ensure they align with operational needs.

Scenario #252: Insufficient Pod Security Policies Leading to Privilege Escalation

- Category: Security

- Environment: K8s v1.21, AWS EKS

  Scenario

- Summary: Insufficiently restrictive PodSecurityPolicies (PSPs) allowed the deployment of privileged pods, which were later exploited by attackers.

- What Happened:A cluster had PodSecurityPolicies enabled, but the policies were too permissive, allowing containers with root privileges and host network access. Attackers exploited these permissions to escalate privileges within the cluster.

- Diagnosis Steps:

  • Checked the PodSecurityPolicy settings and found that they allowed privileged pods and host network access.

  • Identified compromised pods that had root access and were able to communicate freely with other sensitive resources in the cluster.

  Root Cause: Misconfigured PodSecurityPolicy allowed unsafe pods to be deployed with excessive privileges.

- Fix/Workaround:

  • Updated PodSecurityPolicies to enforce stricter controls, such as disallowing privileged containers and restricting host network access.

  • Applied RBAC restrictions to limit who could deploy privileged pods.

- Lessons Learned:

  It is crucial to configure PodSecurityPolicies with the least privilege principle to prevent privilege escalation.

- How to Avoid:

  • Use strict PodSecurityPolicies to enforce safe configurations for all pod deployments.

  • Regularly audit pod configurations and PodSecurityPolicy settings to ensure compliance with security standards.

Scenario #253: Exposed Service Account Token in Pod

- Category: Security

- Environment: K8s v1.20, On-Premise

Scenario

- Summary: A service account token was mistakenly exposed in a pod, allowing attackers to gain unauthorized access to the Kubernetes API.

- What Happened:A developer mistakenly included the service account token in a pod environment variable, making it accessible to anyone with access to the pod. The token was then exploited by attackers to gain unauthorized access to the Kubernetes API.

- Diagnosis Steps:

• Inspected the pod configuration and identified that the service account token was stored in an environment variable.

• Monitored the API server logs and detected unauthorized API calls using the exposed token.

Root Cause: Service account token was inadvertently exposed in the pod's environment variables, allowing attackers to use it for unauthorized access.

- Fix/Workaround:

• Removed the service account token from the environment variable and stored it in a more secure location (e.g., as a Kubernetes Secret).

• Reissued the service account token and rotated the credentials to mitigate potential risks.

- Lessons Learned:

Never expose sensitive credentials like service account tokens in environment variables or in pod specs.

- How to Avoid:

• Store sensitive data, such as service account tokens, in secure locations (Secrets).

• Regularly audit pod configurations to ensure no sensitive information is exposed.

Scenario #254: Rogue Container Executing Malicious Code

- Category: Security

- Environment: K8s v1.22, Azure AKS

Scenario

- Summary: A compromised container running a known exploit executed malicious code that allowed the attacker to gain access to the underlying node.

- What Happened:A container running an outdated image with known vulnerabilities was exploited. The attacker used this vulnerability to gain access to the underlying node and execute malicious commands.

- Diagnosis Steps:

• Conducted a forensic investigation and found that a container was running an outdated image with an unpatched exploit.

• Detected that the attacker used this vulnerability to escape the container and execute commands on the node.

Root Cause: Running containers with outdated or unpatched images introduced security vulnerabilities.

- Fix/Workaround:

• Updated the container images to the latest versions with security patches.

• Implemented automatic image scanning and vulnerability scanning as part of the CI/CD pipeline to catch outdated images before deployment.

- Lessons Learned:

Regularly update container images and scan for vulnerabilities to reduce the attack surface.

- How to Avoid:

• Implement automated image scanning tools to identify vulnerabilities before deploying containers.

• Enforce policies to only allow trusted and updated images to be used in production.

Scenario #255: Overly Permissive Network Policies Allowing Lateral Movement

- Category: Security

- Environment: K8s v1.19, Google Cloud

Scenario

- Summary: Network policies were not restrictive enough, allowing compromised pods to move laterally across the cluster and access other services.

- What Happened:The lack of restrictive network policies allowed any pod to communicate with any other pod in the cluster, even sensitive ones. After a pod was compromised, the attacker moved laterally to other pods and services, leading to further compromise.

- Diagnosis Steps:

• Reviewed the network policy configurations and found that no network isolation was enforced between pods.

• Conducted a post-compromise analysis and found that the attacker moved across multiple services without restriction.

Root Cause: Insufficient network policies allowed unrestricted traffic between pods, increasing the potential for lateral movement.

- Fix/Workaround:

• Implemented restrictive network policies to segment the cluster and restrict traffic between pods based on specific labels and namespaces.

• Ensured that sensitive services were isolated with network policies that only allowed access from trusted sources.

- Lessons Learned:

Strong network segmentation is essential to contain breaches and limit the potential for lateral movement within the cluster.

- How to Avoid:

• Implement and enforce network policies that restrict pod-to-pod communication, especially for sensitive services.

• Regularly audit network policies and adjust them to ensure proper segmentation of workloads.

Scenario #256: Insufficient Encryption for In-Transit Data

- Category: Security

- Environment: K8s v1.23, AWS EKS

Scenario

- Summary: Sensitive data was transmitted in plaintext between services, exposing it to potential eavesdropping and data breaches.

- What Happened:Some internal communications between services in the cluster were not encrypted, which exposed sensitive information during transit. This could have been exploited by attackers using tools to intercept traffic.

- Diagnosis Steps:

• Analyzed service-to-service communication and discovered that some APIs were being called over HTTP rather than HTTPS.

• Monitored network traffic and observed unencrypted data in transit.

Root Cause: Lack of encryption in communication between internal services, resulting in unprotected data being transmitted over the network.

- Fix/Workaround:

  • Configured all services to communicate over HTTPS using TLS encryption.

  • Implemented mutual TLS authentication for all pod-to-pod communications within the cluster.

- Lessons Learned:

  Never allow sensitive data to be transmitted in plaintext across the network. Always enforce encryption.

- How to Avoid:

  • Use Kubernetes network policies to enforce HTTPS communication.

  • Implement and enforce mutual TLS authentication between services.

Scenario #257: Exposing Cluster Services via LoadBalancer with Public IP

- Category: Security

- Environment: K8s v1.21, Google Cloud

  Scenario

- Summary: A service was exposed to the public internet via a LoadBalancer without proper access control, making it vulnerable to attacks.

- What Happened:A service was inadvertently exposed to the internet via an external LoadBalancer, which was not secured. Attackers were able to send requests directly to the service, attempting to exploit vulnerabilities.

- Diagnosis Steps:

  • Inspected the service configuration and found that the type: LoadBalancer was used without any access restrictions.

  • Detected unauthorized attempts to interact with the service from external IPs.

  Root Cause: Misconfiguration allowed the service to be exposed to the public internet without access control.

- Fix/Workaround:

  • Updated the service configuration to use type: ClusterIP or added an appropriate ingress controller with restricted access.

  • Added IP whitelisting or authentication to the exposed services.

- Lessons Learned:

  Always secure services exposed via LoadBalancer by restricting public access or using proper authentication mechanisms.

- How to Avoid:

  • Use ingress controllers with proper access control lists (ACLs) to control inbound traffic.

  • Avoid exposing services unnecessarily; restrict access to only trusted IP ranges.

Scenario #258: Privileged Containers Running Without Seccomp or AppArmor Profiles

- Category: Security

- Environment: K8s v1.20, On-Premise

  Scenario

- Summary: Privileged containers were running without seccomp or AppArmor profiles, leaving the host vulnerable to attacks.

- What Happened:Several containers were deployed with the privileged: true flag, but no seccomp or AppArmor profiles were applied. These containers had unrestricted access to the host kernel, which could lead to security breaches if exploited.

- Diagnosis Steps:

  • Reviewed container configurations and identified containers running with the privileged: true flag.

  • Checked if seccomp or AppArmor profiles were applied and found that none were in place.

  Root Cause: Running privileged containers without applying restrictive security profiles (e.g., seccomp, AppArmor) exposes the host to potential exploitation.

- Fix/Workaround:

  • Disabled the privileged: true flag unless absolutely necessary and applied restrictive seccomp and AppArmor profiles to all privileged containers.

  • Used Kubernetes security policies to prevent the deployment of privileged containers without appropriate security profiles.

- Lessons Learned:

Avoid running containers with excessive privileges. Always apply security profiles to limit the scope of potential attacks.

- How to Avoid:

• Use Kubernetes PodSecurityPolicies (PSPs) or admission controllers to restrict privileged container deployments.

• Enforce the use of seccomp and AppArmor profiles for all containers.

Scenario #259: Malicious Container Image from Untrusted Source

- Category: Security

- Environment: K8s v1.19, Azure AKS

Scenario

- Summary: A malicious container image from an untrusted source was deployed, leading to a security breach in the cluster.

- What Happened:A container image from an untrusted registry was pulled and deployed. The image contained malicious code, which was executed once the container started. The attacker used this to gain unauthorized access to the cluster.

- Diagnosis Steps:

• Analyzed the container image and identified malicious scripts that were executed during the container startup.

• Detected abnormal activity in the cluster, including unauthorized API calls and data exfiltration.

Root Cause: The use of an untrusted container registry allowed the deployment of a malicious container image, which compromised the cluster.

- Fix/Workaround:

• Removed the malicious container image from the cluster and quarantined the affected pods.

• Scanned all images for known vulnerabilities before redeploying containers.

• Configured image admission controllers to only allow images from trusted registries.

- Lessons Learned:

Only use container images from trusted sources, and always scan images for vulnerabilities before deployment.

- How to Avoid:

• Use image signing and validation tools to ensure only trusted images are deployed.

• Implement an image scanning process in the CI/CD pipeline to detect vulnerabilities and malware before deployment.

Scenario #260: Unrestricted Ingress Controller Allowing External Attacks

- Category: Security

- Environment: K8s v1.24, GKE

Scenario

- Summary: The ingress controller was misconfigured, allowing external attackers to bypass network security controls and exploit internal services.

- What Happened:The ingress controller was configured without proper access controls, allowing external users to directly access internal services. Attackers were able to target unprotected services within the cluster.

- Diagnosis Steps:

• Inspected the ingress configuration and found that it was accessible from any IP without authentication.

• Observed attack attempts to access internal services that were supposed to be restricted.

Root Cause: Ingress controller misconfiguration allowed external access to internal services without proper authentication or authorization.

- Fix/Workaround:

• Reconfigured the ingress controller to restrict access to trusted IPs or users via IP whitelisting or authentication.

• Enabled role-based access control (RBAC) to limit access to sensitive services.

- Lessons Learned:

Always configure ingress controllers with proper access control mechanisms to prevent unauthorized access to internal services.

- How to Avoid:

• Use authentication and authorization mechanisms with ingress controllers to protect internal services.

• Regularly audit and update ingress configurations to ensure they align with security policies.

Scenario #261: Misconfigured Ingress Controller Exposing Internal Services

- Category: Security

- Environment: Kubernetes v1.24, GKE

- Summary: An Ingress controller was misconfigured, inadvertently exposing internal services to the public internet.

- What Happened:The default configuration of the Ingress controller allowed all incoming traffic without proper authentication or IP restrictions. This oversight exposed internal services, making them accessible to unauthorized users.

- Diagnosis Steps:

• Reviewed Ingress controller configurations.

• Identified lack of authentication mechanisms and IP whitelisting.

• Detected unauthorized access attempts in logs.

Root Cause: Default Ingress controller settings lacked necessary security configurations.

- Fix/Workaround:

• Implemented IP whitelisting to restrict access.

• Enabled authentication mechanisms for sensitive services.

• Regularly audited Ingress configurations for security compliance.

- Lessons Learned:

Always review and harden default configurations of Ingress controllers to prevent unintended exposure.

- How to Avoid:

• Utilize security best practices when configuring Ingress controllers.

• Regularly audit and update configurations to align with security standards.

Scenario #262: Privileged Containers Without Security Context

- Category: Security

- Environment: Kubernetes v1.22, EKS

- Summary: Containers were running with elevated privileges without defined security contexts, increasing the risk of host compromise.

- What Happened:Several pods were deployed with the privileged: true flag but lacked defined security contexts. This configuration allowed containers to perform operations that could compromise the host system.

- Diagnosis Steps:

• Inspected pod specifications for security context configurations.

• Identified containers running with elevated privileges.

• Assessed potential risks associated with these configurations.

Root Cause: Absence of defined security contexts for privileged containers.

- Fix/Workaround:

• Defined appropriate security contexts for all containers.

• Removed unnecessary privileged access where possible.

• Implemented Pod Security Policies to enforce security standards.

- Lessons Learned:

Clearly define security contexts for all containers, especially those requiring elevated privileges.

- How to Avoid:

• Implement and enforce Pod Security Policies.

• Regularly review and update security contexts for all deployments.

Scenario #263: Unrestricted Network Policies Allowing Lateral Movement

- Category: Security

- Environment: Kubernetes v1.21, Azure AKS

- Summary: Lack of restrictive network policies permitted lateral movement within the cluster after a pod compromise.

- What Happened:An attacker compromised a pod and, due to unrestricted network policies, was able to move laterally within the cluster, accessing other pods and services.

- Diagnosis Steps:

• Reviewed network policy configurations.

• Identified absence of restrictions between pods.

• Traced unauthorized access patterns in network logs.

Root Cause: Inadequate network segmentation due to missing or misconfigured network policies.

- Fix/Workaround:

• Implemented network policies to restrict inter-pod communication.

• Segmented the network based on namespaces and labels.

• Monitored network traffic for unusual patterns.

- Lessons Learned:

Proper network segmentation is crucial to contain breaches and prevent lateral movement.

- How to Avoid:

• Define and enforce strict network policies.

• Regularly audit network configurations and traffic patterns.

Scenario #264: Exposed Kubernetes Dashboard Without Authentication

- Category: Security

- Environment: Kubernetes v1.20, On-Premise

- Summary: The Kubernetes Dashboard was exposed without authentication, allowing unauthorized access to cluster resources.

- What Happened:The Kubernetes Dashboard was deployed with default settings, lacking authentication mechanisms. This oversight allowed anyone with network access to interact with the dashboard and manage cluster resources.

- Diagnosis Steps:

• Accessed the dashboard without credentials.

• Identified the ability to perform administrative actions.

• Checked deployment configurations for authentication settings.

Root Cause: Deployment of the Kubernetes Dashboard without enabling authentication.

- Fix/Workaround:

• Enabled authentication mechanisms for the dashboard.

• Restricted access to the dashboard using network policies.

• Monitored dashboard access logs for unauthorized attempts.

- Lessons Learned:

Always secure administrative interfaces with proper authentication and access controls.

- How to Avoid:

• Implement authentication and authorization for all administrative tools.

• Limit access to management interfaces through network restrictions.

Scenario #265: Use of Vulnerable Container Images

- Category: Security

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Deployment of container images with known vulnerabilities led to potential exploitation risks.

- What Happened:Applications were deployed using outdated container images that contained known vulnerabilities. These vulnerabilities could be exploited by attackers to compromise the application and potentially the cluster.

- Diagnosis Steps:

• Scanned container images for known vulnerabilities.

• Identified outdated packages and unpatched security issues.

• Assessed the potential impact of the identified vulnerabilities.

Root Cause: Use of outdated and vulnerable container images in deployments.

- Fix/Workaround:

• Updated container images to the latest versions with security patches.

• Implemented automated image scanning in the CI/CD pipeline.

• Established a policy to use only trusted and regularly updated images.

- Lessons Learned:

Regularly update and scan container images to mitigate security risks.

- How to Avoid:

• Integrate image scanning tools into the development workflow.

• Maintain an inventory of approved and secure container images.

Scenario #266: Misconfigured Role-Based Access Control (RBAC)

- Category: Security

- Environment: Kubernetes v1.22, GKE

- Summary: Overly permissive RBAC configurations granted users more access than necessary, posing security risks.

- What Happened:Users were assigned roles with broad permissions, allowing them to perform actions beyond their responsibilities. This misconfiguration increased the risk of accidental or malicious changes to the cluster.

- Diagnosis Steps:

  • Reviewed RBAC role and role binding configurations.

  • Identified users with excessive permissions.

  • Assessed the potential impact of the granted permissions.

  Root Cause: Lack of adherence to the principle of least privilege in RBAC configurations.

- Fix/Workaround:

  • Revised RBAC roles to align with user responsibilities.

  • Implemented the principle of least privilege across all roles.

  • Regularly audited RBAC configurations for compliance.

- Lessons Learned:

  Properly configured RBAC is essential to limit access and reduce security risks.

- How to Avoid:

  • Define clear access requirements for each role.

  • Regularly review and update RBAC configurations.

Scenario #267: Insecure Secrets Management

- Category: Security

- Environment: Kubernetes v1.21, On-Premise

- Summary: Secrets were stored in plaintext within configuration files, leading to potential exposure.

- What Happened:Sensitive information, such as API keys and passwords, was stored directly in configuration files without encryption. This practice risked exposure if the files were accessed by unauthorized individuals.

- Diagnosis Steps:

  • Inspected configuration files for embedded secrets.

  • Identified plaintext storage of sensitive information.

  • Evaluated access controls on configuration files.

  Root Cause: Inadequate handling and storage of sensitive information.

- Fix/Workaround:

  • Migrated secrets to Kubernetes Secrets objects.

  • Implemented encryption for secrets at rest and in transit.

  • Restricted access to secrets using RBAC.

- Lessons Learned:

  Proper secrets management is vital to protect sensitive information.

- How to Avoid:

• Use Kubernetes Secrets for managing sensitive data.

• Implement encryption and access controls for secrets.

Scenario #268: Lack of Audit Logging

- Category: Security

- Environment: Kubernetes v1.24, Azure AKS

- Summary: Absence of audit logging hindered the ability to detect and investigate security incidents.

- What Happened:A security incident occurred, but due to the lack of audit logs, it was challenging to trace the actions leading up to the incident and identify the responsible parties.

- Diagnosis Steps:

• Attempted to review audit logs for the incident timeframe.

• Discovered that audit logging was not enabled.

• Assessed the impact of missing audit data on the investigation.

Root Cause: Audit logging was not configured in the Kubernetes cluster.

- Fix/Workaround:

• Enabled audit logging in the cluster.

• Configured log retention and monitoring policies.

• Integrated audit logs with a centralized logging system for analysis.

- Lessons Learned:

Audit logs are essential for monitoring and investigating security events.

- How to Avoid:

• Enable and configure audit logging in all clusters.

• Regularly review and analyze audit logs for anomalies.

Scenario #269: Unrestricted Access to etcd

- Category: Security

- Environment: Kubernetes v1.20, On-Premise

- Summary: The etcd datastore was accessible without authentication, risking exposure of sensitive cluster data.

- What Happened:The etcd service was configured without authentication or encryption, allowing unauthorized users to access and modify cluster state data.

- Diagnosis Steps:

• Attempted to connect to etcd without credentials.

• Successfully accessed sensitive cluster information.

• Evaluated the potential impact of unauthorized access.

Root Cause: Misconfiguration of etcd lacking proper security controls.

- Fix/Workaround:

• Enabled authentication and encryption for etcd.

• Restricted network access to etcd endpoints.

• Regularly audited etcd configurations for security compliance.

- Lessons Learned:

Securing etcd is critical to protect the integrity and confidentiality of cluster data.

- How to Avoid:

• Implement authentication and encryption for etcd.

• Limit access to etcd to authorized personnel and services.

Scenario #270: Absence of Pod Security Policies

- Category: Security

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Without Pod Security Policies, pods were deployed with insecure configurations, increasing the attack surface.

- What Happened:Pods were deployed without restrictions, allowing configurations such as running as root, using host networking, and mounting sensitive host paths, which posed security risks.

- Diagnosis Steps:

  • Reviewed pod specifications for security configurations.

  • Identified insecure settings in multiple deployments.

  • Assessed the potential impact of these configurations.

  Root Cause: Lack of enforced Pod Security Policies to govern pod configurations.

- Fix/Workaround:

  • Implemented Pod Security Policies to enforce security standards.

  • Restricted the use of privileged containers and host resources.

  • Educated development teams on secure pod configurations.

- Lessons Learned:

  Enforcing Pod Security Policies helps maintain a secure and compliant cluster environment.

- How to Avoid:

  • Define and enforce Pod Security Policies.

  • Regularly review pod configurations for adherence to security standards.

Scenario #271: Service Account Token Mounted in All Pods

- Category: Security

- Environment: Kubernetes v1.23, AKS

- Summary: All pods had default service account tokens mounted, increasing the risk of credential leakage.

- What Happened:Developers were unaware that service account tokens were being auto-mounted into every pod, even when not required. If any pod was compromised, its token could be misused to access the Kubernetes API.

- Diagnosis Steps:

  • Inspected pod specs for automountServiceAccountToken.

  • Found all pods had tokens mounted by default.

  • Reviewed logs and discovered unnecessary API calls using those tokens.

  Root Cause: The default behavior of auto-mounting tokens was not overridden.

- Fix/Workaround:

  • Set automountServiceAccountToken: false in non-privileged pods.

  • Reviewed RBAC permissions to ensure tokens were scoped correctly.

- Lessons Learned:

  Don’t give more access than necessary; disable token mounts where not needed.

- How to Avoid:

• Disable token mounting unless required.

• Enforce security-aware pod templates across teams.

Scenario #272: Sensitive Logs Exposed via Centralized Logging

- Category: Security

- Environment: Kubernetes v1.22, EKS with Fluentd

- Summary: Secrets and passwords were accidentally logged and shipped to a centralized logging service accessible to many teams.

- What Happened:Application code logged sensitive values like passwords and access keys, which were picked up by Fluentd and visible in Kibana.

- Diagnosis Steps:

• Reviewed logs after a security audit.

• Discovered multiple log lines with secrets embedded.

• Traced the logs back to specific applications.

Root Cause: Insecure logging practices combined with centralized aggregation.

- Fix/Workaround:

• Removed sensitive logging in app code.

• Configured Fluentd filters to redact secrets.

• Restricted access to sensitive log indices in Kibana.

- Lessons Learned:

Be mindful of what gets logged; logs can become a liability.

- How to Avoid:

• Implement logging best practices.

• Scrub sensitive content before logs leave the app.

Scenario #273: Broken Container Escape Detection

- Category: Security

- Environment: Kubernetes v1.24, GKE

- Summary: A malicious container escaped to host level due to an unpatched kernel, but went undetected due to insufficient monitoring.

- What Happened:A CVE affecting cgroups allowed container breakout. The attacker executed host-level commands and pivoted laterally across nodes.

- Diagnosis Steps:

• Investigated suspicious node-level activity.

• Detected unexpected binaries and processes running as root.

• Correlated with pod logs that had access to /proc.

Root Cause: Outdated host kernel + lack of runtime monitoring.

- Fix/Workaround:

  • Patched all nodes to a secure kernel version.
  • Implemented Falco to monitor syscall anomalies.

- Lessons Learned:

  Container escape is rare but possible—plan for it.

- How to Avoid:

  • Patch host OS regularly.
  • Deploy tools like Falco or Sysdig for anomaly detection.

Scenario #274: Unauthorized Cloud Metadata API Access

- Category: Security

- Environment: Kubernetes v1.22, AWS

- Summary: A pod was able to access the EC2 metadata API and retrieve IAM credentials due to open network access.

- What Happened:A compromised pod accessed the instance metadata service via the default route and used the credentials to access S3 and RDS.

- Diagnosis Steps:

• Analyzed cloudtrail logs for unauthorized S3 access.

• Found requests coming from node metadata credentials.

• Matched with pod's activity timeline.

Root Cause: Lack of egress restrictions from pods to 169.254.169.254.

- Fix/Workaround:

• Restricted pod egress using network policies.

• Enabled IMDSv2 with hop limit = 1 to block pod access.

- Lessons Learned:

Default cloud behaviors can become vulnerabilities in shared nodes.

- How to Avoid:

• Secure instance metadata access.

• Use IRSA (IAM Roles for Service Accounts) instead of node-level credentials.

Scenario #275: Admin Kubeconfig Checked into Git

- Category: Security

- Environment: Kubernetes v1.23, On-Prem

- Summary: A developer accidentally committed a kubeconfig file with full admin access into a public Git repository.

- What Happened:During a code review, a sensitive kubeconfig file was found in a GitHub repo. The credentials allowed full control over the production cluster.

- Diagnosis Steps:

  • Used GitHub search to identify exposed secrets.

  • Retrieved the commit and verified credentials.

  • Checked audit logs for any misuse.

  Root Cause: Lack of .gitignore and secret scanning.

- Fix/Workaround:

  • Rotated the admin credentials immediately.

  • Added secret scanning to CI/CD.

  • Configured .gitignore templates across repos.

- Lessons Learned:

  Accidental leaks happen—monitor and respond quickly.

- How to Avoid:

  • Never store secrets in source code.

  • Use automated secret scanning (e.g., GitHub Advanced Security, TruffleHog).

Scenario #276: JWT Token Replay Attack in Webhook Auth

- Category: Security

- Environment: Kubernetes v1.21, AKS

- Summary: Reused JWT tokens from intercepted API requests were used to impersonate authorized users.

- What Happened:A webhook-based authentication system accepted JWTs without checking their freshness. Tokens were reused in replay attacks.

- Diagnosis Steps:

  • Inspected API server logs for duplicate token use.

  • Found repeated requests with same JWT from different IPs.

  • Correlated with the webhook server not validating expiry/nonce.

  Root Cause: Webhook did not validate tokens properly.

- Fix/Workaround:

  • Updated webhook to validate expiry and nonce in tokens.

  • Rotated keys and invalidated sessions.

- Lessons Learned:

  Token reuse must be considered in authentication systems.

- How to Avoid:

• Use time-limited tokens.

• Implement replay protection with nonces or one-time tokens.

Scenario #277: Container With Hardcoded SSH Keys

- Category: Security

- Environment: Kubernetes v1.20, On-Prem

- Summary: A base image included hardcoded SSH keys which allowed attackers lateral access between environments.

- What Happened:A developer reused a base image with an embedded SSH private key. This key was used across environments and eventually leaked.

- Diagnosis Steps:

• Analyzed image layers with Trivy.

• Found hardcoded private key in /root/.ssh/id_rsa.

• Tested and confirmed it allowed access to multiple systems.

Root Cause: Insecure base image with sensitive files included.

- Fix/Workaround:

• Rebuilt images without sensitive content.

• Rotated all affected SSH keys.

- Lessons Learned:

Never embed sensitive credentials in container images.

- How to Avoid:

  • Scan images before use.

  • Use multistage builds to exclude dev artifacts.

Scenario #278: Insecure Helm Chart Defaults

- Category: Security

- Environment: Kubernetes v1.24, GKE

- Summary: A popular Helm chart had insecure defaults, like exposing dashboards or running as root.

- What Happened:A team installed a chart from a public Helm repo and unknowingly exposed a dashboard on the internet.

- Diagnosis Steps:

  • Discovered open dashboards in a routine scan.

  • Reviewed Helm chart’s default values.

  • Found insecure values.yaml configurations.

Root Cause: Use of Helm chart without overriding insecure defaults.

- Fix/Workaround:

  • Overrode defaults in values.yaml.

  • Audited Helm charts for misconfigurations.

- Lessons Learned:

  Don’t trust defaults—validate every Helm deployment.

- How to Avoid:

  • Read charts carefully before applying.

  • Maintain internal forks of public charts with hardened defaults.

Scenario #279: Shared Cluster with Overlapping Namespaces

- Category: Security

- Environment: Kubernetes v1.22, Shared Dev Cluster

- Summary: Multiple teams used the same namespace naming conventions, causing RBAC overlaps and security concerns.

- What Happened:Two teams created namespaces with the same name across dev environments. RBAC rules overlapped and one team accessed another’s workloads.

- Diagnosis Steps:

• Reviewed RBAC bindings across namespaces.

• Found conflicting roles due to reused namespace names.

• Inspected access logs and verified misuse.

Root Cause: Lack of namespace naming policies in a shared cluster.

- Fix/Workaround:

• Introduced prefix-based namespace naming (e.g., team1-dev).

• Scoped RBAC permissions tightly.

- Lessons Learned:

Namespace naming is security-sensitive in shared clusters.

- How to Avoid:

• Enforce naming policies.

• Use automated namespace creation with templates.

Scenario #280: CVE Ignored in Base Image for Months

- Category: Security

- Environment: Kubernetes v1.23, AWS

- Summary: A known CVE affecting the base image used by multiple services remained unpatched due to no alerting.

- What Happened:A vulnerability in glibc went unnoticed for months because there was no automated CVE scan or alerting. Security only discovered it during a quarterly audit.

- Diagnosis Steps:

  • Scanned container image layers manually.

  • Confirmed multiple CVEs, including critical ones.

  • Traced image origin to a legacy Dockerfile.

  Root Cause: No vulnerability scanning in CI/CD.

- Fix/Workaround:

  • Integrated Clair + Trivy scans into CI/CD pipelines.

  • Setup Slack alerts for critical CVEs.

- Lessons Learned:

  Continuous scanning is vital to security hygiene.

- How to Avoid:

  • Integrate image scanning into build pipelines.

  • Monitor CVE databases for base images regularly.

Scenario #281: Misconfigured PodSecurityPolicy Allowed Privileged Containers

- Category: Security

- Environment: Kubernetes v1.21, On-Prem Cluster

- Summary: Pods were running with privileged: true due to a permissive PodSecurityPolicy (PSP) left enabled during testing.

- What Happened:Developers accidentally left a wide-open PSP in place that allowed privileged containers, host networking, and host path mounts. This allowed a compromised container to access host files.

- Diagnosis Steps:

  • Audited active PSPs.

  • Identified a PSP with overly permissive rules.

  • Found pods using privileged: true.

  Root Cause: Lack of PSP review before production deployment.

- Fix/Workaround:

  • Removed the insecure PSP.

  • Implemented a restrictive default PSP.

  • Migrated to PodSecurityAdmission after PSP deprecation.

- Lessons Learned:

  Security defaults should be restrictive, not permissive.

- How to Avoid:

• Review PSP or PodSecurity configurations regularly.

• Implement strict admission control policies.

Scenario #282: GitLab Runners Spawning Privileged Containers

- Category: Security

- Environment: Kubernetes v1.23, GitLab CI on EKS

- Summary: GitLab runners were configured to run privileged containers to support Docker-in-Docker (DinD), leading to a high-risk setup.

- What Happened:A developer pipeline was hijacked and used to build malicious images, which had access to the underlying node due to privileged mode.

- Diagnosis Steps:

• Detected unusual image pushes to private registry.

• Reviewed runner configuration – found privileged: true enabled.

• Audited node access logs.

Root Cause: Runners configured with elevated privileges for convenience.

- Fix/Workaround:

• Disabled DinD and used Kaniko for builds.

• Set runner securityContext to avoid privilege escalation.

- Lessons Learned:

  Privileged mode should be a last resort.

- How to Avoid:

  • Avoid using DinD where possible.

  • Use rootless build tools like Kaniko or Buildah.

Scenario #283: Kubernetes Secrets Mounted in World-Readable Volumes

- Category: Security

- Environment: Kubernetes v1.24, GKE

- Summary: Secret volumes were mounted with 0644 permissions, allowing any user process inside the container to read them.

- What Happened:A poorly configured application image had other processes running that could access mounted secrets (e.g., service credentials).

- Diagnosis Steps:

  • Reviewed mounted secret volumes and permissions.

  • Identified 0644 file mode on mounted files.

  • Verified multiple processes in the pod could access the secrets.

  Root Cause: Secret volume default mode wasn't overridden.

- Fix/Workaround:

  • Set defaultMode: 0400 on all secret volumes.

  • Isolated processes via containers.

- Lessons Learned:

  Least privilege applies to file access too.

- How to Avoid:

  • Set correct permissions on secret mounts.

  • Use multi-container pods to isolate secrets access.

Scenario #284: Kubelet Port Exposed on Public Interface

- Category: Security

- Environment: Kubernetes v1.20, Bare Metal

- Summary: Kubelet was accidentally exposed on port 10250 to the public internet, allowing unauthenticated metrics and logs access.

- What Happened:Network misconfiguration led to open Kubelet ports without authentication. Attackers scraped pod logs and exploited the /exec endpoint.

- Diagnosis Steps:

• Scanned node ports using nmap.

• Discovered open port 10250 without TLS.

• Verified logs and metrics access externally.

Root Cause: Kubelet served insecure API without proper firewall rules.

- Fix/Workaround:

• Enabled Kubelet authentication and authorization.

• Restricted access via firewall and node security groups.

- Lessons Learned:

Never expose internal components publicly.

- How to Avoid:

• Audit node ports regularly.

• Harden Kubelet with authN/authZ and TLS.

Scenario #285: Cluster Admin Bound to All Authenticated Users

- Category: Security

- Environment: Kubernetes v1.21, AKS

- Summary: A ClusterRoleBinding accidentally granted cluster-admin to all authenticated users due to system:authenticated group.

- What Happened:A misconfigured YAML granted admin access broadly, bypassing intended RBAC restrictions.

- Diagnosis Steps:

  • Audited ClusterRoleBindings.

  • Found binding: subjects: kind: Group, name: system:authenticated.

  • Verified users could create/delete resources cluster-wide.

  Root Cause: RBAC misconfiguration during onboarding automation.

- Fix/Workaround:

  • Deleted the binding immediately.

  • Implemented an RBAC policy validation webhook.

- Lessons Learned:

  Misuse of built-in groups can be catastrophic.

- How to Avoid:

  • Avoid using broad group bindings.

  • Implement pre-commit checks for RBAC files.

Scenario #286: Webhook Authentication Timing Out, Causing Denial of Service

- Category: Security

- Environment: Kubernetes v1.22, EKS

- Summary: Authentication webhook for custom RBAC timed out under load, rejecting valid users and causing cluster-wide issues.

- What Happened:Spike in API requests caused the external webhook server to time out. This led to mass access denials and degraded API server performance.

- Diagnosis Steps:

  • Checked API server logs for webhook timeout messages.

  • Monitored external auth service – saw 5xx errors.

  • Replayed request load to replicate.

  Root Cause: Auth webhook couldn't scale with API server traffic.

- Fix/Workaround:

  • Increased webhook timeouts and horizontal scaling.

  • Added local caching for frequent identities.

- Lessons Learned:

  External dependencies can introduce denial of service risks.

- How to Avoid:

  • Stress-test webhooks.

  • Use token-based or in-cluster auth where possible.

Scenario #287: CSI Driver Exposing Node Secrets

- Category: Security

- Environment: Kubernetes v1.24, CSI Plugin (AWS Secrets Store)

- Summary: Misconfigured CSI driver exposed secrets on hostPath mount accessible to privileged pods.

- What Happened:Secrets mounted via the CSI driver were not isolated properly, allowing another pod with hostPath access to read them.

- Diagnosis Steps:

  • Reviewed CSI driver logs and configurations.

  • Found secrets mounted in shared path (/var/lib/...).

  • Identified privilege escalation path via hostPath.

  Root Cause: CSI driver exposed secrets globally on node filesystem.

- Fix/Workaround:

  • Scoped CSI mounts with per-pod directories.

  • Disabled hostPath access for workloads.

- Lessons Learned:

  CSI drivers must be hardened like apps.

- How to Avoid:

  • Test CSI secrets exposure under threat models.

  • Restrict node-level file access via policies.

Scenario #288: EphemeralContainers Used for Reconnaissance

- Category: Security

- Environment: Kubernetes v1.25, GKE

- Summary: A compromised user deployed ephemeral containers to inspect and copy secrets from running pods.

- What Happened:A user with access to ephemeralcontainers feature spun up containers in critical pods and read mounted secrets and env vars.

- Diagnosis Steps:

  • Audited API server calls to ephemeralcontainers API.

  • Found suspicious container launches.

  • Inspected shell history and accessed secrets.

  Root Cause: Overprivileged user with ephemeralcontainers access.

- Fix/Workaround:

• Removed permissions to ephemeral containers for all roles.

• Set audit policies for their use.

- Lessons Learned:

New features introduce new attack vectors.

- How to Avoid:

• Lock down access to new APIs.

• Monitor audit logs for container injection attempts.

Scenario #289: hostAliases Used for Spoofing Internal Services

- Category: Security

- Environment: Kubernetes v1.22, On-Prem

- Summary: Malicious pod used hostAliases to spoof internal service hostnames and intercept requests.

- What Happened:An insider attack modified /etc/hosts in a pod using hostAliases to redirect requests to attacker-controlled services.

- Diagnosis Steps:

• Reviewed pod manifests with hostAliases.

• Captured outbound DNS traffic and traced redirections.

• Detected communication with rogue internal services.

Root Cause: Abuse of hostAliases field in PodSpec.

- Fix/Workaround:

• Disabled use of hostAliases via OPA policies.

• Logged all pod specs with custom host entries.

- Lessons Learned:

Host file spoofing can bypass DNS-based security.

- How to Avoid:

• Restrict or disallow use of hostAliases.

• Rely on service discovery via DNS only.

Scenario #290: Privilege Escalation via Unchecked securityContext in Helm Chart

- Category: Security

- Environment: Kubernetes v1.21, Helm v3.8

- Summary: A third-party Helm chart allowed setting arbitrary securityContext, letting users run pods as root in production.

- What Happened:A chart exposed securityContext overrides without constraints. A developer added runAsUser: 0during deployment, leading to root-level containers.

- Diagnosis Steps:

• Inspected Helm chart values and rendered manifests.

• Detected containers with runAsUser: 0.

• Reviewed change logs in GitOps pipeline.

Root Cause: Chart did not validate or restrict securityContext fields.

- Fix/Workaround:

• Forked chart and restricted overrides via schema.

• Implemented OPA Gatekeeper to block root containers.

- Lessons Learned:

Helm charts can be as dangerous as code.

- How to Avoid:

• Validate all chart values.

• Use policy engines to restrict risky configurations.

Scenario #291: Service Account Token Leakage via Logs

- Category: Security

- Environment: Kubernetes v1.23, AKS

- Summary: Application inadvertently logged its mounted service account token, exposing it to log aggregation systems.

- What Happened:A misconfigured logging library dumped all environment variables and mounted file contents at startup, including the token from /var/run/secrets/kubernetes.io/serviceaccount/token.

- Diagnosis Steps:

  • Searched central logs for token patterns.

  • Confirmed multiple logs contained valid JWTs.

  • Validated token usage in audit logs.

  Root Cause: Poor logging hygiene in application code.

- Fix/Workaround:

  • Rotated all impacted service account tokens.

  • Added environment and file sanitization to logging library.

- Lessons Learned:

  Tokens are sensitive credentials and should never be logged.

- How to Avoid:

  • Add a startup check to prevent token exposure.

  • Use static analysis or OPA to block risky mounts/logs.

Scenario #292: Escalation via Editable Validating WebhookConfiguration

- Category: Security

- Environment: Kubernetes v1.24, EKS

- Summary: User with edit rights on a validating webhook modified it to bypass critical security policies.

- What Happened:An internal user reconfigured the webhook to always return allow, disabling cluster-wide security checks.

- Diagnosis Steps:

  • Detected anomaly: privileged pods getting deployed.

  • Checked webhook configuration history in GitOps.

  • Verified that failurePolicy: Ignore and static allow logic were added.

  Root Cause: Lack of control over webhook configuration permissions.

- Fix/Workaround:

  • Restricted access to ValidatingWebhookConfiguration objects.

  • Added checksums to webhook definitions in GitOps.

- Lessons Learned:

  Webhooks must be tightly controlled to preserve cluster security.

- How to Avoid:

• Lock down RBAC access to webhook configurations.

• Monitor changes with alerts and diff checks.

Scenario #293: Stale Node Certificates After Rejoining Cluster

- Category: Security

- Environment: Kubernetes v1.21, Kubeadm-based cluster

- Summary: A node was rejoined to the cluster using a stale certificate, giving it access it shouldn't have.

- What Happened:A node that was previously removed was added back using an old /var/lib/kubelet/pki/kubelet-client.crt, which was still valid.

- Diagnosis Steps:

• Compared certificate expiry and usage.

• Found stale kubelet cert on rejoined node.

• Verified node had been deleted previously.

Root Cause: Old credentials not purged before node rejoin.

- Fix/Workaround:

• Manually deleted old certificates from the node.

• Set short TTLs for client certificates.

- Lessons Learned:

Node certs should be one-time-use and short-lived.

- How to Avoid:

• Rotate node credentials regularly.

• Use automation to purge sensitive files before rejoining.

Scenario #294: ArgoCD Exploit via Unverified Helm Charts

- Category: Security

- Environment: Kubernetes v1.24, ArgoCD

- Summary: ArgoCD deployed a malicious Helm chart that added privileged pods and container escape backdoors.

- What Happened:A team added a new Helm repo that wasn't verified. The chart had post-install hooks that ran containers with host access.

- Diagnosis Steps:

• Found unusual pods using hostNetwork and hostPID.

• Traced deployment to ArgoCD application with external chart.

• Inspected chart source – found embedded malicious hooks.

Root Cause: Lack of chart verification or provenance checks.

- Fix/Workaround:

  • Removed the chart and all related workloads.
  • Enabled Helm OCI signatures and repo allow-lists.

- Lessons Learned:

  Supply chain security is critical, even with GitOps.

- How to Avoid:

  • Only use verified or internal Helm repos.
  • Enable ArgoCD Helm signature verification.

Scenario #295: Node Compromise via Insecure Container Runtime

- Category: Security

- Environment: Kubernetes v1.22, CRI-O on Bare Metal

- Summary: A CVE in the container runtime allowed a container breakout, leading to full node compromise.

- What Happened:An attacker exploited CRI-O vulnerability (CVE-2022-0811) that allowed containers to overwrite host paths via sysctl injection.

- Diagnosis Steps:

• Detected abnormal node CPU spike and external traffic.

• Inspected containers – found sysctl modifications.

• Cross-verified with known CVEs.

Root Cause: Unpatched CRI-O vulnerability and default seccomp profile disabled.

- Fix/Workaround:

• Upgraded CRI-O to patched version.

• Enabled seccomp and AppArmor by default.

- Lessons Learned:

Container runtimes must be hardened and patched like any system component.

- How to Avoid:

• Automate CVE scanning for runtime components.

• Harden runtimes with security profiles.

Scenario #296: Workload with Wildcard RBAC Access to All Secrets

- Category: Security

- Environment: Kubernetes v1.23, Self-Hosted

- Summary: A microservice was granted get and list access to all secrets cluster-wide using *.

- What Happened:Developers gave overly broad access to a namespace-wide controller, leading to accidental exposure of unrelated team secrets.

- Diagnosis Steps:

  • Audited RBAC for secrets access.

  • Found RoleBinding with resources: [“secrets”], verbs: [“get”, “list”], resourceNames: ["*"].

  Root Cause: Overly broad RBAC permissions in service manifest.

- Fix/Workaround:

  • Replaced wildcard permissions with explicit named secrets.

  • Enabled audit logging on all secrets API calls.

- Lessons Learned:

* in RBAC is often overkill and dangerous.

- How to Avoid:

  • Use least privilege principle.

  • Validate RBAC via CI/CD linting tools.

Scenario #297: Malicious Init Container Used for Reconnaissance

- Category: Security

- Environment: Kubernetes v1.25, GKE

- Summary: A pod was launched with a benign main container and a malicious init container that copied node metadata.

- What Happened:Init container wrote node files (e.g., /etc/resolv.conf, cloud instance metadata) to an external bucket before terminating.

- Diagnosis Steps:

  • Enabled audit logs for object storage.

  • Traced writes back to a pod with suspicious init container.

  • Reviewed init container image – found embedded exfil logic.

  Root Cause: Lack of validation on init container behavior.

- Fix/Workaround:

  • Blocked unknown container registries via policy.

  • Implemented runtime security agents to inspect init behavior.

- Lessons Learned:

  Init containers must be treated as full-fledged security risks.

- How to Avoid:

• Verify init container images and registries.

• Use runtime tools (e.g., Falco) for behavior analysis.

Scenario #298: Ingress Controller Exposed /metrics Without Auth

- Category: Security

- Environment: Kubernetes v1.24, NGINX Ingress

- Summary: Prometheus scraping endpoint /metrics was exposed without authentication and revealed sensitive internal details.

- What Happened:A misconfigured ingress rule allowed external users to access /metrics, which included upstream paths, response codes, and error logs.

- Diagnosis Steps:

• Scanned public URLs.

• Found /metrics exposed to unauthenticated traffic.

• Inspected NGINX ingress annotations.

Root Cause: Ingress annotations missing auth and whitelist rules.

- Fix/Workaround:

• Applied IP whitelist and basic auth for /metrics.

• Added network policies to restrict access.

- Lessons Learned:

  Even observability endpoints need protection.

- How to Avoid:

  • Enforce auth for all public endpoints.

  • Separate internal vs. external monitoring targets.

Scenario #299: Secret Stored in ConfigMap by Mistake

- Category: Security

- Environment: Kubernetes v1.23, AKS

- Summary: A sensitive API key was accidentally stored in a ConfigMap instead of a Secret, making it visible in plain text.

- What Happened:Developer used a ConfigMap for application config, and mistakenly included an apiKey in it. Anyone with view rights could read it.

- Diagnosis Steps:

  • Reviewed config files for plaintext secrets.

  • Found hardcoded credentials in ConfigMap YAML.

  Root Cause: Misunderstanding of Secret vs. ConfigMap usage.

- Fix/Workaround:

• Moved key to a Kubernetes Secret.

• Rotated exposed credentials.

- Lessons Learned:

Educate developers on proper resource usage.

- How to Avoid:

• Lint manifests to block secrets in ConfigMaps.

• Train developers in security best practices.

Scenario #300: Token Reuse After Namespace Deletion and Recreation

- Category: Security

- Environment: Kubernetes v1.24, Self-Hosted

- Summary: A previously deleted namespace was recreated, and old tokens (from backups) were still valid and worked.

- What Happened:Developer restored a backup including secrets from a deleted namespace. The token was still valid and allowed access to cluster resources.

- Diagnosis Steps:

• Found access via old token in logs.

• Verified namespace was deleted, then recreated with same name.

• Checked secrets in restored backup.

Root Cause: Static tokens persisted after deletion and recreation.

- Fix/Workaround:

• Rotated all tokens after backup restore.

• Implemented TTL-based token policies.

- Lessons Learned:

Tokens must be invalidated after deletion or restore.

- How to Avoid:

• Don’t restore old secrets blindly.

• Rotate and re-issue credentials post-restore.

4. Storage

Scenario #301: PVC Stuck in Terminating State After Node Crash

- Category: Storage

- Environment: Kubernetes v1.22, EBS CSI Driver on EKS

- Summary: A node crash caused a PersistentVolumeClaim (PVC) to be stuck in Terminating, blocking pod deletion.

- What Happened:The node hosting the pod with the PVC crashed and never returned. The volume was still attached, and Kubernetes couldn't cleanly unmount or delete it.

- Diagnosis Steps:

  • Described the PVC: status was Terminating.

  • Checked finalizers on the PVC object.

  • Verified the volume was still attached to the crashed node via AWS Console.

  Root Cause: The volume attachment record wasn't cleaned up due to the ungraceful node failure.

- Fix/Workaround:

  • Manually removed the PVC finalizers.

  • Used aws ec2 detach-volume to forcibly detach.

- Lessons Learned:

  Finalizers can block PVC deletion in edge cases.

- How to Avoid:

  • Use the external-attacher CSI sidecar with leader election.

  • Implement automation to detect and clean up stuck attachments.

Scenario #302: Data Corruption on HostPath Volumes

- Category: Storage

- Environment: Kubernetes v1.20, Bare Metal

- Summary: Multiple pods sharing a HostPath volume led to inconsistent file states and eventual corruption.

- What Happened:Two pods were writing to the same HostPath volume concurrently, which wasn't designed for concurrent write access. Files became corrupted due to race conditions.

- Diagnosis Steps:

  • Identified common HostPath mount across pods.
  • Checked application logs — showed file write conflicts.
  • Inspected corrupted data on disk.

  Root Cause: Lack of coordination and access control on shared HostPath.

- Fix/Workaround:

  • Moved workloads to CSI-backed volumes with ReadWriteOnce enforcement.
  • Ensured only one pod accessed a volume at a time.

- Lessons Learned:

  HostPath volumes offer no isolation or locking guarantees.

- How to Avoid:

• Use CSI volumes with enforced access modes.

• Avoid HostPath unless absolutely necessary.

Scenario #303: Volume Mount Fails Due to Node Affinity Mismatch

- Category: Storage

- Environment: Kubernetes v1.23, GCE PD on GKE

- Summary: A pod was scheduled on a node that couldn't access the persistent disk due to zone mismatch.

- What Happened:A StatefulSet PVC was bound to a disk in us-central1-a, but the pod got scheduled in us-central1-b, causing volume mount failure.

- Diagnosis Steps:

• Described pod: showed MountVolume.MountDevice failed.

• Described PVC and PV: zone mismatch confirmed.

• Looked at scheduler decisions — no awareness of volume zone.

Root Cause: Scheduler was unaware of zone constraints on the PV.

- Fix/Workaround:

• Added topology.kubernetes.io/zone node affinity to match PV.

• Ensured StatefulSets used storage classes with volume binding mode WaitForFirstConsumer.

- Lessons Learned:

Without delayed binding, PVs can bind in zones that don’t match future pods.

- How to Avoid:

• Use WaitForFirstConsumer for dynamic provisioning.

• Always define zone-aware topology constraints.

Scenario #304: PVC Not Rescheduled After Node Deletion

- Category: Storage

- Environment: Kubernetes v1.21, Azure Disk CSI

- Summary: A StatefulSet pod failed to reschedule after its node was deleted, due to Azure disk still being attached.

- What Happened:A pod using Azure Disk was on a node that was manually deleted. Azure did not automatically detach the disk, so rescheduling failed.

- Diagnosis Steps:

• Pod stuck in ContainerCreating.

• CSI logs showed "Volume is still attached to another node".

• Azure Portal confirmed volume was attached.

Root Cause: Manual node deletion bypassed volume detachment logic.

- Fix/Workaround:

• Detached the disk from the Azure console.

• Recreated pod successfully on another node.

- Lessons Learned:

Manual infrastructure changes can break Kubernetes assumptions.

- How to Avoid:

• Use automation/scripts for safe node draining and deletion.

• Monitor CSI detachment status on node removal.

Scenario #305: Long PVC Rebinding Time on StatefulSet Restart

- Category: Storage

- Environment: Kubernetes v1.24, Rook Ceph

- Summary: Restarting a StatefulSet with many PVCs caused long downtime due to slow rebinding.

- What Happened:A 20-replica StatefulSet was restarted, and each pod waited for its PVC to rebind and attach. Ceph mount operations were sequential and slow.

- Diagnosis Steps:

  • Pods stuck at Init stage for 15–20 minutes.

  • Ceph logs showed delayed attachment per volume.

  • Described PVCs: bound but not mounted.

  Root Cause: Sequential volume mount throttling and inefficient CSI attach policies.

- Fix/Workaround:

  • Tuned CSI attach concurrency.

  • Split the StatefulSet into smaller chunks.

- Lessons Learned:

  Large-scale StatefulSets need volume attach tuning.

- How to Avoid:

  • Parallelize pod restarts using partitioned rollouts.

  • Monitor CSI mount throughput.

Scenario #306: CSI Volume Plugin Crash Loops Due to Secret Rotation

- Category: Storage

- Environment: Kubernetes v1.25, Vault CSI Provider

- Summary: Volume plugin entered crash loop after secret provider's token was rotated unexpectedly.

- What Happened:A service account used by the Vault CSI plugin had its token rotated mid-operation. The plugin couldn't fetch new credentials and crashed.

- Diagnosis Steps:

 • CrashLoopBackOff on csi-vault-provider pods.

 • Logs showed "401 Unauthorized" from Vault.

 • Verified service account token changed recently.

 Root Cause: No logic in plugin to handle token change or re-auth.

- Fix/Workaround:

 • Restarted the CSI plugin pods.

 • Upgraded plugin to a version with token refresh logic.

- Lessons Learned:

 CSI providers must gracefully handle credential rotations.

- How to Avoid:

 • Use projected service account tokens with auto-refresh.

 • Monitor plugin health on secret rotations.

Scenario #307: ReadWriteMany PVCs Cause IO Bottlenecks

- Category: Storage

- Environment: Kubernetes v1.23, NFS-backed PVCs

- Summary: Heavy read/write on a shared PVC caused file IO contention and throttling across pods.

- What Happened:Multiple pods used a shared ReadWriteMany PVC for scratch space. Concurrent writes led to massive IO wait times and high pod latency.

- Diagnosis Steps:

  • High pod latency and CPU idle time.

  • Checked NFS server: high disk and network usage.

  • Application logs showed timeouts.

  Root Cause: No coordination or locking on shared writable volume.

- Fix/Workaround:

  • Partitioned workloads to use isolated volumes.

  • Added cache layer for reads.

- Lessons Learned:

  RWX volumes are not always suitable for concurrent writes.

- How to Avoid:

• Use RWX volumes for read-shared data only.

• Avoid writes unless using clustered filesystems (e.g., CephFS).

Scenario #308: PVC Mount Timeout Due to PodSecurityPolicy

- Category: Storage

- Environment: Kubernetes v1.21, PSP Enabled Cluster

- Summary: A pod couldn't mount a volume because PodSecurityPolicy (PSP) rejected required fsGroup.

- What Happened:A storage class required fsGroup for volume mount permissions. The pod didn't set it, and PSP disallowed dynamic group assignment.

- Diagnosis Steps:

• Pod stuck in CreateContainerConfigError.

• Events showed "pod rejected by PSP".

• Storage class required fsGroup.

Root Cause: Incompatible PSP with volume mount security requirements.

- Fix/Workaround:

• Modified PSP to allow required fsGroup range.

• Updated pod security context.

- Lessons Learned:

  Storage plugins often need security context alignment.

- How to Avoid:

  • Review storage class requirements.

  • Align security policies with volume specs.

Scenario #309: Orphaned PVs After Namespace Deletion

- Category: Storage

- Environment: Kubernetes v1.20, Self-Hosted

- Summary: Deleting a namespace did not clean up PersistentVolumes, leading to leaked storage.

- What Happened:A team deleted a namespace with PVCs, but the associated PVs (with Retain policy) remained and weren't cleaned up.

- Diagnosis Steps:

  • Listed all PVs: found orphaned volumes in Released state.

  • Checked reclaim policy: Retain.

  Root Cause: Manual cleanup required for Retain policy.

- Fix/Workaround:

• Deleted old PVs and disks manually.

• Changed reclaim policy to Delete for dynamic volumes.

- Lessons Learned:

Reclaim policy should match cleanup expectations.

- How to Avoid:

• Use Delete unless you need manual volume recovery.

• Monitor Released PVs for leaks.

Scenario #310: StorageClass Misconfiguration Blocks Dynamic Provisioning

- Category: Storage

- Environment: Kubernetes v1.25, GKE

- Summary: New PVCs failed to bind due to a broken default StorageClass with incorrect parameters.

- What Happened:A recent update modified the default StorageClass to use a non-existent disk type. All PVCs created with default settings failed provisioning.

- Diagnosis Steps:

• PVCs in Pending state.

• Checked events: “failed to provision volume with StorageClass”.

• Described StorageClass: invalid parameter type: ssd2.

Root Cause: Mistyped disk type in StorageClass definition.

- Fix/Workaround:

• Corrected StorageClass parameters.

• Manually bound PVCs with valid classes.

- Lessons Learned:

Default StorageClass affects many workloads.

- How to Avoid:

• Validate StorageClass on cluster upgrades.

• Use automated tests for provisioning paths.

Scenario #311: StatefulSet Volume Cloning Results in Data Leakage

- Category: Storage

- Environment: Kubernetes v1.24, CSI Volume Cloning enabled

- Summary: Cloning PVCs between StatefulSet pods led to shared data unexpectedly appearing in new replicas.

- What Happened:Engineers used volume cloning to duplicate data for new pods. They assumed data would be copied and isolated. However, clones preserved file locks and session metadata, which caused apps to behave erratically.

- Diagnosis Steps:

  • New pods accessed old session data unexpectedly.

  • lsblk and md5sum on cloned volumes showed identical data.

  • Verified cloning was done via StorageClass that didn't support true snapshot isolation.

  Root Cause: Misunderstanding of cloning behavior — logical clone ≠ deep copy.

- Fix/Workaround:

  • Stopped cloning and switched to backup/restore-based provisioning.

  • Used rsync with integrity checks instead.

- Lessons Learned:

  Not all clones are deep copies; understand your CSI plugin's clone semantics.

- How to Avoid:

  • Use cloning only for stateless data unless supported thoroughly.

  • Validate cloned volume content before production use.

Scenario #312: Volume Resize Not Reflected in Mounted Filesystem

- Category: Storage

- Environment: Kubernetes v1.22, OpenEBS

- Summary: Volume expansion was successful on the PV, but pods didn't see the increased space.

- What Happened:After increasing PVC size, the PV reflected the new size, but df -h inside the pod still showed the old size.

- Diagnosis Steps:

  • Checked PVC and PV: showed expanded size.

  • Pod logs indicated no disk space.

  • mount inside pod showed volume was mounted but not resized.

  Root Cause: Filesystem resize not triggered automatically.

- Fix/Workaround:

  • Restarted pod to remount the volume and trigger resize.

  • Verified resize2fs logs in CSI driver.

- Lessons Learned:

  Volume resizing may require pod restarts depending on CSI driver.

- How to Avoid:

  • Schedule a rolling restart after volume resize operations.

  • Check if your CSI driver supports online filesystem resizing.

Scenario #313: CSI Controller Pod Crash Due to Log Overflow

- Category: Storage

- Environment: Kubernetes v1.23, Longhorn

- Summary: The CSI controller crashed repeatedly due to unbounded logging filling up ephemeral storage.

- What Happened:A looped RPC error generated thousands of log lines per second. Node /var/log/containers hit 100% disk usage.

- Diagnosis Steps:

  • kubectl describe pod: showed OOMKilled and failed to write logs.
  • Checked node disk: /var was full.
  • Logs rotated too slowly.

  Root Cause: Verbose logging + missing log throttling + small disk.

- Fix/Workaround:

  • Added log rate limits via CSI plugin config.
  • Increased node ephemeral storage.

- Lessons Learned:

  Logging misconfigurations can become outages.

- How to Avoid:

  • Monitor log volume and disk usage.

  • Use log rotation and retention policies.

Scenario #314: PVs Stuck in Released Due to Missing Finalizer Removal

- Category: Storage

- Environment: Kubernetes v1.21, NFS

- Summary: PVCs were deleted, but PVs remained stuck in Released, preventing reuse.

- What Happened:PVC deletion left behind PVs marked as Released, and the NFS driver didn't remove finalizers, blocking clean-up.

- Diagnosis Steps:

  • Listed PVs: showed Released, with kubernetes.io/pv-protection finalizer still present.

  • Couldn't bind new PVCs due to status: Released.

  Root Cause: Driver didn't implement Delete reclaim logic properly.

- Fix/Workaround:

  • Patched PVs to remove finalizers.

• Recycled or deleted volumes manually.

- Lessons Learned:

Some drivers require manual cleanup unless fully CSI-compliant.

- How to Avoid:

• Use CSI drivers with full lifecycle support.

• Monitor PV statuses regularly.

Scenario #315: CSI Driver DaemonSet Deployment Missing Tolerations for Taints

- Category: Storage

- Environment: Kubernetes v1.25, Bare Metal

- Summary: CSI Node plugin DaemonSet didn’t deploy on all nodes due to missing taint tolerations.

- What Happened:Storage nodes were tainted (node-role.kubernetes.io/storage:NoSchedule), and the CSI DaemonSet didn’t tolerate it, so pods failed to mount volumes.

- Diagnosis Steps:

• CSI node pods not scheduled on certain nodes.

• Checked node taints vs DaemonSet tolerations.

• Pods stuck in Pending.

Root Cause: Taint/toleration mismatch in CSI node plugin manifest.

- Fix/Workaround:

• Added required tolerations to DaemonSet.

- Lessons Learned:

Storage plugins must tolerate relevant node taints to function correctly.

- How to Avoid:

• Review node taints and CSI tolerations during setup.

• Use node affinity and tolerations for critical system components.

Scenario #316: Mount Propagation Issues with Sidecar Containers

- Category: Storage

- Environment: Kubernetes v1.22, GKE

- Summary: Sidecar containers didn't see mounted volumes due to incorrect mountPropagation settings.

- What Happened:An app container wrote to a mounted path, but sidecar container couldn't read the changes.

- Diagnosis Steps:

  • Logs in sidecar showed empty directory.

  • Checked volumeMounts: missing mountPropagation: Bidirectional.

  Root Cause: Default mount propagation is None, blocking volume visibility between containers.

- Fix/Workaround:

  • Added mountPropagation: Bidirectional to shared volumeMounts.

- Lessons Learned:

  Without correct propagation, shared volumes don't work across containers.

- How to Avoid:

  • Understand container mount namespaces.

  • Always define propagation when using shared mounts.

Scenario #317: File Permissions Reset on Pod Restart

- Category: Storage

- Environment: Kubernetes v1.20, CephFS

- Summary: Pod volume permissions reset after each restart, breaking application logic.

- What Happened:App wrote files with specific UID/GID. After restart, files were inaccessible due to CephFS resetting ownership.

- Diagnosis Steps:

  • Compared ls -l before/after restart.

  • Storage class used fsGroup: 9999 by default.

  Root Cause: PodSecurityContext didn't override fsGroup, so default applied every time.

- Fix/Workaround:

  • Set explicit securityContext.fsGroup in pod spec.

- Lessons Learned:

  CSI plugins may enforce ownership unless overridden.

- How to Avoid:

  • Always declare expected ownership with securityContext.

Scenario #318: Volume Mount Succeeds but Application Can't Write

- Category: Storage

- Environment: Kubernetes v1.23, EBS

- Summary: Volume mounted correctly, but application failed to write due to filesystem mismatch.

- What Happened:App expected xfs but volume formatted as ext4. Some operations silently failed or corrupted.

- Diagnosis Steps:

  • Application logs showed invalid argument on file ops.

  • CSI driver defaulted to ext4.

  • Verified with df -T.

  Root Cause: Application compatibility issue with default filesystem.

- Fix/Workaround:

  • Used storage class parameter to specify xfs.

- Lessons Learned:

  Filesystem types matter for certain workloads.

- How to Avoid:

  • Align volume formatting with application expectations.

Scenario #319: Volume Snapshot Restore Includes Corrupt Data

- Category: Storage

- Environment: Kubernetes v1.24, Velero + CSI Snapshots

- Summary: Snapshot-based restore brought back corrupted state due to hot snapshot timing.

- What Happened:Velero snapshot was taken during active write burst. Filesystem was inconsistent at time of snapshot.

- Diagnosis Steps:

  • App logs showed corrupted files after restore.

  • Snapshot logs showed no quiescing.

  • Restore replayed same state.

  Root Cause: No pre-freeze or app-level quiescing before snapshot.

- Fix/Workaround:

  • Paused writes before snapshot.

  • Enabled filesystem freeze hook in Velero plugin.

- Lessons Learned:

  Snapshots must be coordinated with app state.

- How to Avoid:

  • Use pre/post hooks for consistent snapshotting.

Scenario #320: Zombie Volumes Occupying Cloud Quota

- Category: Storage

- Environment: Kubernetes v1.25, AWS EBS

- Summary: Deleted PVCs didn't release volumes due to failed detach steps, leading to quota exhaustion.

- What Happened:PVCs were deleted, but EBS volumes stayed in-use, blocking provisioning of new ones due to quota limits.

- Diagnosis Steps:

  • Checked AWS Console: volumes remained.
  • Described events: detach errors during node crash.

  Root Cause: CSI driver missed final detach due to abrupt node termination.

- Fix/Workaround:

  • Manually detached and deleted volumes.
  • Adjusted controller retry limits.

- Lessons Learned:

  Cloud volumes may silently linger even after PVC/PV deletion.

- How to Avoid:

• Use cloud resource monitoring.

• Add alerts for orphaned volumes.

Scenario #321: Volume Snapshot Garbage Collection Fails

- Category: Storage

- Environment: Kubernetes v1.25, CSI Snapshotter with Velero

- Summary: Volume snapshots piled up because snapshot objects were not getting garbage collected after use.

- What Happened:Snapshots triggered via Velero remained in the cluster even after restore, eventually exhausting cloud snapshot limits and storage quota.

- Diagnosis Steps:

• Listed all VolumeSnapshots and VolumeSnapshotContents — saw hundreds still in ReadyToUse: true state.

• Checked finalizers on snapshot objects — found snapshot.storage.kubernetes.io/volumesnapshot not removed.

• Velero logs showed successful restore but no cleanup action.

Root Cause: Snapshot GC controller didn't remove finalizers due to missing permissions in Velero's service account.

- Fix/Workaround:

- Added required RBAC rules to Velero.
- Manually deleted stale snapshot objects.

- Lessons Learned:

Improperly configured snapshot permissions can stall GC.

- How to Avoid:

- Always test snapshot and restore flows end-to-end.
- Enable automated cleanup in your backup tooling.

Scenario #322: Volume Mount Delays Due to Node Drain Stale Attachment

- Category: Storage

- Environment: Kubernetes v1.23, AWS EBS CSI

- Summary: Volumes took too long to attach on new nodes after pod rescheduling due to stale attachment metadata.

- What Happened:After draining a node for maintenance, workloads failed over, but volume attachments still pointed to old node, causing delays in remount.

- Diagnosis Steps:

- Described PV: still had attachedNode as drained one.

• Cloud logs showed volume in-use errors.

• CSI controller didn't retry detach fast enough.

Root Cause: Controller had exponential backoff on detach retries.

- Fix/Workaround:

• Reduced backoff limit in CSI controller config.

• Used manual detach via cloud CLI in emergencies.

- Lessons Learned:

Volume operations can get stuck in edge-node cases.

- How to Avoid:

• Use health checks to ensure detach success before draining.

• Monitor VolumeAttachment objects during node ops.

Scenario #323: Application Writes Lost After Node Reboot

- Category: Storage

- Environment: Kubernetes v1.21, Local Persistent Volumes

- Summary: After a node reboot, pod restarted, but wrote to a different volume path, resulting in apparent data loss.

- What Happened:Application data wasn’t persisted after a power cycle because the mount point dynamically changed.

- Diagnosis Steps:

• Compared volume paths before and after reboot.

• Found PV had hostPath mount with no stable binding.

• Volume wasn’t pinned to specific disk partition.

Root Cause: Local PV was defined with generic hostPath, not using local volume plugin with device references.

- Fix/Workaround:

• Refactored PV to use local with nodeAffinity.

• Explicitly mounted disk partitions.

- Lessons Learned:

hostPath should not be used for production data.

- How to Avoid:

• Always use local storage plugin for node-local disks.

• Avoid loosely defined persistent paths.

Scenario #324: Pod CrashLoop Due to Read-Only Volume Remount

- Category: Storage

- Environment: Kubernetes v1.22, GCP Filestore

- Summary: Pod volume was remounted as read-only after a transient network disconnect, breaking app write logic.

- What Happened:During a brief NFS outage, volume was remounted in read-only mode by the NFS client. Application kept crashing due to inability to write logs.

- Diagnosis Steps:

  • Checked mount logs: showed NFS remounted as read-only.

  • kubectl describe pod: showed volume still mounted.

  • Pod logs: permission denied on write.

  Root Cause: NFS client behavior defaults to remount as read-only after timeout.

- Fix/Workaround:

  • Restarted pod to trigger clean remount.

  • Tuned NFS mount options (soft, timeo, retry).

- Lessons Learned:

  NFS remount behavior can silently switch access mode.

- How to Avoid:

  • Monitor for dmesg or NFS client remounts.

  • Add alerts for unexpected read-only volume transitions.

Scenario #325: Data Corruption on Shared Volume With Two Pods

- Category: Storage

- Environment: Kubernetes v1.23, NFS PVC shared by 2 pods

- Summary: Two pods writing to the same volume caused inconsistent files and data loss.

- What Happened:Both pods ran jobs writing to the same output files. Without file locking, one pod overwrote data from the other.

- Diagnosis Steps:

  • Logs showed incomplete file writes.

  • File hashes changed mid-run.

  • No mutual exclusion mechanism implemented.

  Root Cause: Shared volume used without locking or coordination between pods.

- Fix/Workaround:

  • Refactored app logic to coordinate file writes via leader election.

  • Used a queue-based processing system.

- Lessons Learned:

  Shared volume access must be controlled explicitly.

- How to Avoid:

  • Never assume coordination when using shared volumes.
  • Use per-pod PVCs or job-level locking.

Scenario #326: Mount Volume Exceeded Timeout

- Category: Storage

- Environment: Kubernetes v1.26, Azure Disk CSI

- Summary: Pod remained stuck in ContainerCreating state because volume mount operations timed out.

- What Happened:CSI node plugin had stale cache and attempted mount on incorrect device path. Retry logic delayed pod start by ~15 minutes.

- Diagnosis Steps:

  • Described pod: stuck with Unable to mount volume error.
  • Node CSI logs: device not found.
  • Saw old mount references in plugin cache.

  Root Cause: Plugin did not invalidate mount state properly after a failed mount.

- Fix/Workaround:

  • Cleared plugin cache manually.

• Upgraded CSI driver to fixed version.

- Lessons Learned:

CSI drivers can introduce delays through stale state.

- How to Avoid:

• Keep CSI drivers up-to-date.

• Use pre-mount checks to validate device paths.

Scenario #327: Static PV Bound to Wrong PVC

- Category: Storage

- Environment: Kubernetes v1.21, Manually created PVs

- Summary: A misconfigured static PV got bound to the wrong PVC, exposing sensitive data.

- What Happened:Two PVCs had overlapping selectors. The PV intended for app-A was bound to app-B, which accessed restricted files.

- Diagnosis Steps:

• Checked PV annotations: saw wrong PVC UID.

• File system showed app-A data.

• Both PVCs used identical storageClassName and no selector.

Root Cause: Ambiguous PV selection caused unintended binding.

- Fix/Workaround:

  • Used volumeName field in PVCs for direct binding.
  • Set explicit labels/selectors to isolate.

- Lessons Learned:

  Manual PVs require strict binding rules.

- How to Avoid:

  • Use volumeName for static PV binding.
  • Avoid reusing storageClassName across different apps.

Scenario #328: Pod Eviction Due to DiskPressure Despite PVC

- Category: Storage

- Environment: Kubernetes v1.22, Local PVs

- Summary: Node evicted pods due to DiskPressure, even though app used dedicated PVC backed by a separate disk.

- What Happened:Node root disk filled up with log data, triggering eviction manager. The PVC itself was healthy and not full.

- Diagnosis Steps:

  • Node describe: showed DiskPressure condition true.

  • Application pod evicted due to node pressure, not volume pressure.

  • Root disk had full /var/log.

  Root Cause: Kubelet doesn’t distinguish between root disk and attached volumes for eviction triggers.

- Fix/Workaround:

  • Cleaned logs from root disk.

  • Moved logging to PVC-backed location.

- Lessons Learned:

  PVCs don’t protect from node-level disk pressure.

- How to Avoid:

  • Monitor node root disks in addition to volume usage.

  • Redirect logs and temp files to PVCs.

Scenario #329: Pod Gets Stuck Due to Ghost Mount Point

- Category: Storage

- Environment: Kubernetes v1.20, iSCSI volumes

- Summary: Pod failed to start because the mount point was partially deleted, leaving the system confused.

- What Happened:After node crash, the iSCSI mount folder remained but device wasn’t attached. New pod couldn’t proceed due to leftover mount artifacts.

- Diagnosis Steps:

  • CSI logs: mount path exists but not a mount point.

  • mount | grep iscsi — returned nothing.

  • ls /mnt/... — folder existed with empty contents.

  Root Cause: Stale mount folder confused CSI plugin logic.

- Fix/Workaround:

  • Manually deleted stale mount folders.

  • Restarted kubelet on affected node.

- Lessons Learned:

  Mount lifecycle must be cleanly managed.

- How to Avoid:

  • Use pre-start hooks to validate mount point integrity.

  • Include cleanup logic in custom CSI deployments.

Scenario #330: PVC Resize Broke StatefulSet Ordering

- Category: Storage

- Environment: Kubernetes v1.24, StatefulSets + RWO PVCs

- Summary: When resizing PVCs, StatefulSet pods restarted in parallel, violating ordinal guarantees.

- What Happened:PVC expansion triggered pod restarts, but multiple pods came up simultaneously, causing database quorum failures.

- Diagnosis Steps:

 • Checked StatefulSet controller behavior — PVC resize didn't preserve pod startup order.

 • App logs: quorum could not be established.

 Root Cause: StatefulSet controller didn't serialize PVC resizes.

- Fix/Workaround:

 • Manually controlled pod restarts during PVC resize.

 • Added readiness gates to enforce sequential boot.

- Lessons Learned:

 StatefulSets don't coordinate PVC changes well.

- How to Avoid:

• Use podManagementPolicy: OrderedReady.

• Handle resizes during maintenance windows.

Scenario #331: ReadAfterWrite Inconsistency on Object Store-Backed CSI

- Category: Storage

- Environment: Kubernetes v1.26, MinIO CSI driver, Ceph RGW backend

- Summary: Applications experienced stale reads immediately after writing to the same file via CSI mount backed by an S3-like object store.

- What Happened:A distributed app wrote metadata and then read it back to validate—however, the file content was outdated due to eventual consistency in object backend.

- Diagnosis Steps:

• Logged file hashes before and after write — mismatch seen.

• Found underlying storage was S3-compatible with eventual consistency.

• CSI driver buffered writes asynchronously.

Root Cause: Object store semantics (eventual consistency) not suitable for synchronous read-after-write patterns.

- Fix/Workaround:

• Introduced write barriers and retry logic in app.

• Switched to CephFS for strong consistency.

- Lessons Learned:

  Object store-backed volumes need strong consistency guards.

- How to Avoid:

  • Avoid using S3-style backends for workloads expecting POSIX semantics.

  • Use CephFS, NFS, or block storage for transactional I/O.

Scenario #332: PV Resize Fails After Node Reboot

- Category: Storage

- Environment: Kubernetes v1.24, AWS EBS

- Summary: After a node reboot, a PVC resize request remained pending, blocking pod start.

- What Happened:VolumeExpansion was triggered via PVC patch. But after a node reboot, controller couldn't find the in-use mount point to complete fsResize.

- Diagnosis Steps:

  • PVC status.conditions showed FileSystemResizePending.

  • CSI node plugin logs showed missing device.

  • Node reboot removed mount references prematurely.

Root Cause: Resize operation depends on volume being mounted at the time of filesystem expansion.

- Fix/Workaround:

  • Reattached volume by starting pod temporarily on the node.
  • Resize completed automatically.

- Lessons Learned:

  Filesystem resize requires node readiness and volume mount.

- How to Avoid:

  • Schedule resizes during stable node windows.
  • Use pvc-resize readiness gates in automation.

Scenario #333: CSI Driver Crash Loops on VolumeAttach

- Category: Storage

- Environment: Kubernetes v1.22, OpenEBS Jiva CSI

- Summary: CSI node plugin entered CrashLoopBackOff due to panic during volume attach, halting all storage provisioning.

- What Happened:VolumeAttachment object triggered a plugin bug—CSI crashed during RPC call, making storage class unusable.

- Diagnosis Steps:

  • Checked CSI node logs — Go panic in attach handler.

  • Pods using Jiva SC failed with AttachVolume.Attach failed error.

  • CSI pod restarted every few seconds.

  Root Cause: Volume metadata had an unexpected field due to version mismatch.

- Fix/Workaround:

  • Rolled back CSI driver to stable version.

  • Purged corrupted volume metadata.

- Lessons Learned:

  CSI versioning must be tightly managed.

- How to Avoid:

  • Use upgrade staging before deploying new CSI versions.

  • Enable CSI health monitoring via liveness probes.

Scenario #334: PVC Binding Fails Due to Multiple Default StorageClasses

- Category: Storage

- Environment: Kubernetes v1.23

- Summary: PVC creation failed intermittently because the cluster had two storage classes marked as default.

- What Happened:Two different teams installed their storage plugins (EBS and Rook), both marked default. PVC binding randomly chose one.

- Diagnosis Steps:

  • Ran kubectl get storageclass — two entries with is-default-class=true.

  • PVCs had no storageClassName, leading to random binding.

  • One SC used unsupported reclaimPolicy.

  Root Cause: Multiple default StorageClasses confuse the scheduler.

- Fix/Workaround:

  • Patched one SC to remove the default annotation.

  • Explicitly specified SC in Helm charts.

- Lessons Learned:

  Default SC conflicts silently break provisioning.

- How to Avoid:

  • Enforce single default SC via cluster policy.

  • Always specify storageClassName explicitly in critical apps.

Scenario #335: Zombie VolumeAttachment Blocks New PVC

- Category: Storage

- Environment: Kubernetes v1.21, Longhorn

- Summary: After a node crash, a VolumeAttachment object was not garbage collected, blocking new PVCs from attaching.

- What Happened:Application tried to use the volume, but Longhorn saw the old attachment from a dead node and refused reattachment.

- Diagnosis Steps:

 • Listed VolumeAttachment resources — found one pointing to a non-existent node.

 • Longhorn logs: volume already attached to another node.

 • Node was removed forcefully.

 Root Cause: VolumeAttachment controller did not clean up orphaned entries on node deletion.

- Fix/Workaround:

 • Manually deleted VolumeAttachment.

 • Restarted CSI pods to refresh state.

- Lessons Learned:

 Controller garbage collection is fragile post-node failure.

- How to Avoid:

• Use node lifecycle hooks to detach volumes gracefully.

• Alert on dangling VolumeAttachments.

Scenario #336: Persistent Volume Bound But Not Mounted

- Category: Storage

- Environment: Kubernetes v1.25, NFS

- Summary: Pod entered Running state, but data was missing because PV was bound but not properly mounted.

- What Happened:NFS server was unreachable during pod start. Pod started, but mount failed silently due to default retry behavior.

- Diagnosis Steps:

• mount output lacked NFS entry.

• Pod logs: No such file or directory errors.

• CSI logs showed silent NFS timeout.

Root Cause: CSI driver didn’t fail pod start when mount failed.

- Fix/Workaround:

• Added mountOptions: [hard,intr] to NFS SC.

• Set pod readiness probe to check file existence.

- Lessons Learned:

  Mount failures don’t always stop pod startup.

- How to Avoid:

  • Validate mounts via init containers or probes.

  • Monitor CSI logs on pod lifecycle events.

Scenario #337: CSI Snapshot Restore Overwrites Active Data

- Category: Storage

- Environment: Kubernetes v1.26, CSI snapshots (v1beta1)

- Summary: User triggered a snapshot restore to an existing PVC, unintentionally overwriting live data.

- What Happened:Snapshot restore process recreated PVC from source but didn't prevent overwriting an already-mounted volume.

- Diagnosis Steps:

  • Traced VolumeSnapshotContent and PVC references.

  • PVC had reclaimPolicy: Retain, but was reused.

  • SnapshotClass used Delete policy.

  Root Cause: No validation existed between snapshot restore and in-use PVCs.

- Fix/Workaround:

  • Restored snapshot to a new PVC and used manual copy/move.

  • Added lifecycle checks before invoking restores.

- Lessons Learned:

  Restoring snapshots can be destructive.

- How to Avoid:

  • Never restore to in-use PVCs without backup.

  • Build snapshot workflows that validate PVC state.

Scenario #338: Incomplete Volume Detach Breaks Node Scheduling

- Category: Storage

- Environment: Kubernetes v1.22, iSCSI

- Summary: Scheduler skipped a healthy node due to a ghost VolumeAttachment that was never cleaned up.

- What Happened:Node marked as ready, but volume controller skipped scheduling new pods due to “in-use” flag on volumes from a deleted pod.

- Diagnosis Steps:

• Described unscheduled pod — failed to bind due to volume already attached.

• VolumeAttachment still referenced old pod.

• CSI logs showed no detach command received.

Root Cause: CSI controller restart dropped detach request queue.

- Fix/Workaround:

• Recreated CSI controller pod.

• Requeued detach operation via manual deletion.

- Lessons Learned:

CSI recovery from mid-state crash is critical.

- How to Avoid:

• Persist attach/detach queues.

• Use cloud-level health checks for cleanup.

Scenario #339: App Breaks Due to Missing SubPath After Volume Expansion

- Category: Storage

- Environment: Kubernetes v1.24, PVC with subPath

- Summary: After PVC expansion, the mount inside pod pointed to root of volume, not the expected subPath.

- What Happened:Application was configured to mount /data/subdir. After resizing, pod restarted, and subPath was ignored, mounting full volume at /data.

- Diagnosis Steps:

  • Pod logs showed missing directory structure.
  • Inspected pod spec: subPath was correct.
  • CSI logs: subPath expansion failed due to permissions.

  Root Cause: CSI driver did not remap subPath after resize correctly.

- Fix/Workaround:

  • Changed pod to recreate the subPath explicitly.
  • Waited for bugfix release from CSI provider.

- Lessons Learned:

  PVC expansion may break subPath unless handled explicitly.

- How to Avoid:

  • Avoid complex subPath usage unless tested under all lifecycle events.
  • Watch CSI release notes carefully.

Scenario #340: Backup Restore Process Created Orphaned PVCs

- Category: Storage

- Environment: Kubernetes v1.23, Velero

- Summary: A namespace restore from backup recreated PVCs that had no matching PVs, blocking further deployment.

- What Happened:Velero restored PVCs without matching spec.volumeName. Since PVs weren't backed up, they remained Pending.

- Diagnosis Steps:

  • PVC status showed Pending, with no bound PV.

  • Described PVC: no volumeName, no SC.

  • Velero logs: skipped PV restore due to config.

  Root Cause: Restore policy did not include PVs.

- Fix/Workaround:

  • Recreated PVCs manually with correct storage class.

  • Re-enabled PV backup in Velero settings.

- Lessons Learned:

  Partial restores break PVC-PV binding logic.

- How to Avoid:

  • Always back up PVs with PVCs in stateful applications.

• Validate restore completeness before deployment.

Scenario #341: Cross-Zone Volume Binding Fails with StatefulSet

- Category: Storage

- Environment: Kubernetes v1.25, AWS EBS, StatefulSet with anti-affinity

- Summary: Pods in a StatefulSet failed to start due to volume binding constraints when spread across zones.

- What Happened:Each pod had a PVC, but volumes couldn’t be bound because the preferred zones didn't match pod scheduling constraints.

- Diagnosis Steps:

• Pod events: failed to provision volume with StorageClass "gp2" due to zone mismatch.

• kubectl describe pvc showed Pending.

• StorageClass had allowedTopologies defined, conflicting with affinity rules.

Root Cause: StatefulSet pods with zone anti-affinity clashed with single-zone EBS volume provisioning.

- Fix/Workaround:

• Updated StorageClass to allow all zones.

• Aligned affinity rules with allowed topologies.

- Lessons Learned:

  StatefulSets and volume topology must be explicitly aligned.

- How to Avoid:

  • Use multi-zone-aware volume plugins like EFS or FSx when spreading pods.

Scenario #342: Volume Snapshot Controller Race Condition

- Category: Storage

- Environment: Kubernetes v1.23, CSI Snapshot Controller

- Summary: Rapid creation/deletion of snapshots caused the controller to panic due to race conditions in snapshot finalizers.

- What Happened:Automation created/deleted hundreds of snapshots per minute. The controller panicked due to concurrent finalizer modifications.

- Diagnosis Steps:

  • Observed controller crash loop in logs.

  • Snapshot objects stuck in Terminating state.

  • Controller logs: resourceVersion conflict.

  Root Cause: Finalizer updates not serialized under high load.

- Fix/Workaround:

  • Throttled snapshot requests.

  • Patched controller deployment to limit concurrency.

- Lessons Learned:

  High snapshot churn breaks stability.

- How to Avoid:

  • Monitor snapshot queue metrics.

  • Apply rate limits in CI/CD snapshot tests.

Scenario #343: Failed Volume Resize Blocks Rollout

- Category: Storage

- Environment: Kubernetes v1.24, CSI VolumeExpansion enabled

- Summary: Deployment rollout got stuck because one of the pods couldn’t start due to a failed volume expansion.

- What Happened:Admin updated PVC to request more storage. Resize failed due to volume driver limitation. New pods remained in Pending.

- Diagnosis Steps:

• PVC events: resize not supported for current volume type.

• Pod events: volume resize pending.

Root Cause: Underlying CSI driver didn't support in-use resize.

- Fix/Workaround:

• Deleted affected pods, allowed volume to unmount.

• Resize succeeded offline.

- Lessons Learned:

Not all CSI drivers handle online expansion.

- How to Avoid:

• Check CSI driver support for in-use expansion.

• Add pre-checks before resizing PVCs.

Scenario #344: Application Data Lost After Node Eviction

- Category: Storage

- Environment: Kubernetes v1.23, hostPath volumes

- Summary: Node drained for maintenance led to permanent data loss for apps using hostPath volumes.

- What Happened:Stateful workloads were evicted. When pods rescheduled on new nodes, the volume path was empty.

- Diagnosis Steps:

  • Observed empty application directories post-scheduling.

  • Confirmed hostPath location was not shared across nodes.

  Root Cause: hostPath volumes are node-specific and not portable.

- Fix/Workaround:

  • Migrated to CSI-based dynamic provisioning.

  • Used NFS for shared storage.

- Lessons Learned:

  hostPath is unsafe for stateful production apps.

- How to Avoid:

  • Use portable CSI drivers for persistent data.

  • Restrict hostPath usage with admission controllers.

Scenario #345: Read-Only PV Caused Write Failures After Restore

- Category: Storage

- Environment: Kubernetes v1.22, Velero, AWS EBS

- Summary: After restoring from backup, the volume was attached as read-only, causing application crashes.

- What Happened:Backup included PVCs and PVs, but not associated VolumeAttachment states. Restore marked volume read-only to avoid conflicts.

- Diagnosis Steps:

  • Pod logs: permission denied on writes.

  • PVC events: attached in read-only mode.

  • AWS console showed volume attachment flag.

  Root Cause: Velero restored volumes without resetting VolumeAttachment mode.

- Fix/Workaround:

  • Detached and reattached the volume manually as read-write.

  • Updated Velero plugin to handle VolumeAttachment explicitly.

- Lessons Learned:

  Restores need to preserve attachment metadata.

- How to Avoid:

  • Validate post-restore PVC/PV attachment states.

  • Use snapshot/restore plugins that track attachment mode.

Scenario #346: NFS Server Restart Crashes Pods

- Category: Storage

- Environment: Kubernetes v1.24, in-cluster NFS server

- Summary: NFS server restarted for upgrade. All dependent pods crashed due to stale file handles and unmount errors.

- What Happened:NFS mount became stale after server restart. Pods using volumes got stuck in crash loops.

- Diagnosis Steps:

  • Pod logs: Stale file handle, I/O error.

  • Kernel logs showed NFS timeout.

  Root Cause: NFS state is not stateless across server restarts unless configured.

- Fix/Workaround:

  • Enabled NFSv4 stateless mode.

  • Recovered pods by restarting them post-reboot.

- Lessons Learned:

  In-cluster storage servers need HA design.

- How to Avoid:

• Use managed NFS services or replicated storage.

• Add pod liveness checks for filesystem readiness.

Scenario #347: VolumeBindingBlocked Condition Causes Pod Scheduling Delay

- Category: Storage

- Environment: Kubernetes v1.25, dynamic provisioning

- Summary: Scheduler skipped over pods with pending PVCs due to VolumeBindingBlocked status, even though volumes were eventually created.

- What Happened:PVC triggered provisioning, but until PV was available, pod scheduling was deferred.

- Diagnosis Steps:

• Pod condition: PodScheduled: False, reason VolumeBindingBlocked.

• StorageClass had delayed provisioning.

• PVC was Pending for ~60s.

Root Cause: Volume provisioning time exceeded scheduling delay threshold.

- Fix/Workaround:

• Increased controller timeout thresholds.

• Optimized provisioning backend latency.

- Lessons Learned:

Storage latency can delay workloads unexpectedly.

- How to Avoid:

  • Monitor PVC creation latency in Prometheus.
  • Use pre-created PVCs for latency-sensitive apps.

Scenario #348: Data Corruption from Overprovisioned Thin Volumes

- Category: Storage

- Environment: Kubernetes v1.22, LVM-CSI thin provisioning

- Summary: Under heavy load, pods reported data corruption. Storage layer had thinly provisioned LVM volumes that overcommitted disk.

- What Happened:Thin pool ran out of physical space during write bursts, leading to partial writes and corrupted files.

- Diagnosis Steps:

  • Pod logs: checksum mismatches.
  • Node logs: thin pool out of space.
  • LVM command showed 100% usage.

Root Cause: Thin provisioning wasn't monitored and exceeded safe limits.

- Fix/Workaround:

  • Increased physical volume backing the pool.

  • Set strict overcommit alerting.

- Lessons Learned:

  Thin provisioning is risky under unpredictable loads.

- How to Avoid:

  • Monitor usage with lvdisplay, dmsetup.

  • Avoid thin pools in production without full monitoring.

Scenario #349: VolumeProvisioningFailure on GKE Due to IAM Misconfiguration

- Category: Storage

- Environment: GKE, Workload Identity enabled

- Summary: CSI driver failed to provision new volumes due to missing IAM permissions, even though StorageClass was valid.

- What Happened:GCP Persistent Disk CSI driver couldn't create disks because the service account lacked compute permissions.

- Diagnosis Steps:

• Event logs: failed to provision volume with StorageClass: permission denied.

• IAM policy lacked compute.disks.create.

Root Cause: CSI driver operated under workload identity with incorrect bindings.

- Fix/Workaround:

• Granted missing IAM permissions to the bound service account.

• Restarted CSI controller.

- Lessons Learned:

IAM and CSI need constant alignment in cloud environments.

- How to Avoid:

• Use pre-flight IAM checks during cluster provisioning.

• Bind GKE Workload Identity properly.

Scenario #350: Node Crash Triggers Volume Remount Loop

- Category: Storage

- Environment: Kubernetes v1.26, CSI, NVMes

- Summary: After a node crash, volume remount loop occurred due to conflicting device paths.

- What Happened:Volume had a static device path cached in CSI driver. Upon node recovery, OS assigned a new device path. CSI couldn't reconcile.

- Diagnosis Steps:

  • CSI logs: device path not found.

  • Pod remained in ContainerCreating.

  • OS showed volume present under different path.

  Root Cause: CSI assumed static device path, OS changed it post-reboot.

- Fix/Workaround:

  • Added udev rules for consistent device naming.

  • Restarted CSI daemon to detect new device path.

- Lessons Learned:

  Relying on device paths can break persistence.

- How to Avoid:

  • Use device UUIDs or filesystem labels where supported.

  • Restart CSI pods post-reboot events.

Scenario #351: VolumeMount Conflict Between Init and Main Containers

- Category: Storage

- Environment: Kubernetes v1.25, containerized database restore job

- Summary: Init container and main container used the same volume path but with different modes, causing the main container to crash.

- What Happened:An init container wrote a backup file to a shared volume. The main container expected a clean mount, found conflicting content, and failed on startup.

- Diagnosis Steps:

  • Pod logs showed file already exists error.

  • Examined pod manifest: both containers used the same volumeMount.path.

  Root Cause: Shared volume path caused file conflicts between lifecycle stages.

- Fix/Workaround:

  • Used a subPath for the init container to isolate file writes.

  • Moved backup logic to an external init job.

- Lessons Learned:

  Volume sharing across containers must be carefully scoped.

- How to Avoid:

  • Always use subPath if write behavior differs.

  • Isolate volume use per container stage when possible.

Scenario #352: PVCs Stuck in “Terminating” Due to Finalizers

- Category: Storage

- Environment: Kubernetes v1.24, CSI driver with finalizer

- Summary: After deleting PVCs, they remained in Terminating state indefinitely due to stuck finalizers.

- What Happened:The CSI driver responsible for finalizer cleanup was crash-looping, preventing PVC finalizer execution.

- Diagnosis Steps:

  • PVCs had finalizer external-attacher.csi.driver.io.

  • CSI pod logs showed repeated panics due to malformed config.

  Root Cause: Driver bug prevented cleanup logic, blocking PVC deletion.

- Fix/Workaround:

  • Patched the driver deployment.

  • Manually removed finalizers using kubectl patch.

- Lessons Learned:

  CSI finalizer bugs can block resource lifecycle.

- How to Avoid:

• Regularly update CSI drivers.

• Monitor PVC lifecycle duration metrics.

Scenario #353: Misconfigured ReadOnlyMany Mount Blocks Write Operations

- Category: Storage

- Environment: Kubernetes v1.23, NFS volume

- Summary: Volume mounted as ReadOnlyMany blocked necessary write operations, despite NFS server allowing writes.

- What Happened:VolumeMount was incorrectly marked as readOnly: true. Application failed on write attempts.

- Diagnosis Steps:

• Application logs: read-only filesystem.

• Pod manifest showed readOnly: true.

Root Cause: Misconfiguration in the volumeMounts spec.

- Fix/Workaround:

• Updated the manifest to readOnly: false.

- Lessons Learned:

Read-only flags silently break expected behavior.

- How to Avoid:

  • Validate volume mount flags in CI.

  • Use initContainer to test mount behavior.

Scenario #354: In-Tree Plugin PVs Lost After Driver Migration

- Category: Storage

- Environment: Kubernetes v1.26, in-tree to CSI migration

- Summary: Existing in-tree volumes became unrecognized after enabling CSI migration.

- What Happened:Migrated GCE volumes to CSI plugin. Old PVs had legacy annotations and didn't bind correctly.

- Diagnosis Steps:

  • PVs showed Unavailable state.

  • Migration feature gates enabled but missing annotations.

  Root Cause: Backward incompatibility in migration logic for pre-existing PVs.

- Fix/Workaround:

  • Manually edited PV annotations to match CSI requirements.

- Lessons Learned:

  Migration feature gates must be tested in staging.

- How to Avoid:

  • Run migration with shadow mode first.

  • Migrate PVs gradually using tools like pv-migrate.

Scenario #355: Pod Deleted but Volume Still Mounted on Node

- Category: Storage

- Environment: Kubernetes v1.24, CSI

- Summary: Pod was force-deleted, but its volume wasn't unmounted from the node, blocking future pod scheduling.

- What Happened:Force deletion bypassed CSI driver cleanup. Mount lingered and failed future pod volume attach.

- Diagnosis Steps:

  • kubectl describe node showed volume still attached.

  • lsblk confirmed mount on node.

  • Logs showed attach errors.

  Root Cause: Orphaned mount due to force deletion.

- Fix/Workaround:

  • Manually unmounted the volume on node.

  • Drained and rebooted the node.

- Lessons Learned:

  Forced pod deletions should be last resort.

- How to Avoid:

  • Set up automated orphaned mount detection scripts.

  • Use graceful deletion with finalizer handling.

Scenario #356: Ceph RBD Volume Crashes Pods Under IOPS Saturation

- Category: Storage

- Environment: Kubernetes v1.23, Ceph CSI

- Summary: Under heavy I/O, Ceph volumes became unresponsive, leading to kernel-level I/O errors in pods.

- What Happened:Application workload created sustained random writes. Ceph cluster's IOPS limit was reached.

- Diagnosis Steps:

• dmesg logs: blk_update_request: I/O error.

• Pod logs: database fsync errors.

• Ceph health: HEALTH_WARN: slow ops.

Root Cause: Ceph RBD pool under-provisioned for the workload.

- Fix/Workaround:

• Migrated to SSD-backed Ceph pools.

• Throttled application concurrency.

- Lessons Learned:

Distributed storage systems fail silently under stress.

- How to Avoid:

• Benchmark storage before rollout.

• Alert on high RBD latency.

Scenario #357: ReplicaSet Using PVCs Fails Due to VolumeClaimTemplate Misuse

- Category: Storage

- Environment: Kubernetes v1.25

- Summary: Developer tried using volumeClaimTemplates in a ReplicaSet manifest, which isn't supported.

- What Happened:Deployment applied, but pods failed to create PVCs.

- Diagnosis Steps:

  • Controller logs: volumeClaimTemplates is not supported in ReplicaSet.

  • No PVCs appeared in kubectl get pvc.

  Root Cause: volumeClaimTemplates is only supported in StatefulSet.

- Fix/Workaround:

  • Refactored ReplicaSet to StatefulSet.

- Lessons Learned:

  Not all workload types support dynamic PVCs.

- How to Avoid:

  • Use workload reference charts during manifest authoring.

  • Validate manifests with policy engines like OPA.

Scenario #358: Filesystem Type Mismatch During Volume Attach

- Category: Storage

- Environment: Kubernetes v1.24, ext4 vs xfs

- Summary: A pod failed to start because the PV expected ext4 but the node formatted it as xfs.

- What Happened:Pre-provisioned disk had xfs, but StorageClass defaulted to ext4.

- Diagnosis Steps:

  • Attach logs: mount failed: wrong fs type.

  • blkid on node showed xfs.

  Root Cause: Filesystem mismatch between PV and node assumptions.

- Fix/Workaround:

  • Reformatted disk to ext4.

  • Aligned StorageClass with PV fsType.

- Lessons Learned:

  Filesystem types must match across the stack.

- How to Avoid:

  • Explicitly set fsType in StorageClass.

  • Document provisioner formatting logic.

Scenario #359: iSCSI Volumes Fail After Node Kernel Upgrade

- Category: Storage

- Environment: Kubernetes v1.26, CSI iSCSI plugin

- Summary: Post-upgrade, all pods using iSCSI volumes failed to mount due to kernel module incompatibility.

- What Happened:Kernel upgrade removed or broke iscsi_tcp module needed by CSI driver.

- Diagnosis Steps:

  • CSI logs: no such device iscsi_tcp.

  • modprobe iscsi_tcp failed.

  • Pod events: mount timeout.

  Root Cause: Node image didn't include required kernel modules post-upgrade.

- Fix/Workaround:

  • Installed open-iscsi and related modules.

  • Rebooted node.

- Lessons Learned:

  OS updates can break CSI compatibility.

- How to Avoid:

• Pin node kernel versions.

• Run upgrade simulations in canary clusters.

Scenario #360: PVs Not Deleted After PVC Cleanup Due to Retain Policy

- Category: Storage

- Environment: Kubernetes v1.23, AWS EBS

- Summary: After PVCs were deleted, underlying PVs and disks remained, leading to cloud resource sprawl.

- What Happened:Retain policy on the PV preserved the disk after PVC was deleted.

- Diagnosis Steps:

• kubectl get pv showed status Released.

• Disk still visible in AWS console.

Root Cause: PV reclaimPolicy was Retain, not Delete.

- Fix/Workaround:

• Manually deleted PVs and EBS volumes.

- Lessons Learned:

Retain policy needs operational follow-up.

- How to Avoid:

  • Use Delete policy unless manual cleanup is required.
  • Audit dangling PVs regularly.

Scenario #361: Concurrent Pod Scheduling on the Same PVC Causes Mount Conflict

- Category: Storage

- Environment: Kubernetes v1.24, AWS EBS, ReadWriteOnce PVC

- Summary: Two pods attempted to use the same PVC simultaneously, causing one pod to be stuck in ContainerCreating.

- What Happened:A deployment scale-up triggered duplicate pods trying to mount the same EBS volume on different nodes.

- Diagnosis Steps:

  • One pod was running, the other stuck in ContainerCreating.
  • Events showed Volume is already attached to another node.

  Root Cause: EBS supports ReadWriteOnce, not multi-node attach.

- Fix/Workaround:

• Added anti-affinity to restrict pod scheduling to a single node.

• Used EFS (ReadWriteMany) for workloads needing shared storage.

- Lessons Learned:

Not all storage supports multi-node access.

- How to Avoid:

• Understand volume access modes.

• Use StatefulSets or anti-affinity for PVC sharing.

Scenario #362: StatefulSet Pod Replacement Fails Due to PVC Retention

- Category: Storage

- Environment: Kubernetes v1.23, StatefulSet with volumeClaimTemplates

- Summary: Deleted a StatefulSet pod manually, but new pod failed due to existing PVC conflict.

- What Happened:PVC persisted after pod deletion due to StatefulSet retention policy.

- Diagnosis Steps:

• kubectl get pvc showed PVC still bound.

• New pod stuck in Pending.

Root Cause: StatefulSet retains PVCs unless explicitly deleted.

- Fix/Workaround:

  • Deleted old PVC manually to let StatefulSet recreate it.

- Lessons Learned:

  Stateful PVCs are tightly coupled to pod identity.

- How to Avoid:

  • Use persistentVolumeReclaimPolicy: Delete only when data can be lost.

  • Automate cleanup for failed StatefulSet replacements.

Scenario #363: HostPath Volume Access Leaks Host Data into Container

- Category: Storage

- Environment: Kubernetes v1.22, single-node dev cluster

- Summary: HostPath volume mounted the wrong directory, exposing sensitive host data to the container.

- What Happened:Misconfigured path / instead of /data allowed container full read access to host.

- Diagnosis Steps:

• Container listed host files under /mnt/host.

• Pod manifest showed path: /.

Root Cause: Typo in the volume path.

- Fix/Workaround:

• Corrected volume path in manifest.

• Revoked pod access.

- Lessons Learned:

HostPath has minimal safety nets.

- How to Avoid:

• Avoid using HostPath unless absolutely necessary.

• Validate mount paths through automated policies.

Scenario #364: CSI Driver Crashes When Node Resource Is Deleted Prematurely

- Category: Storage

- Environment: Kubernetes v1.25, custom CSI driver

- Summary: Deleting a node object before the CSI driver detached volumes caused crash loops.

- What Happened:Admin manually deleted a node before volume detach completed.

- Diagnosis Steps:

  • CSI logs showed panic due to missing node metadata.

  • Pods remained in Terminating.

  Root Cause: Driver attempted to clean up mounts from a non-existent node resource.

- Fix/Workaround:

  • Waited for CSI driver to timeout and self-recover.

  • Rebooted node to forcibly detach volumes.

- Lessons Learned:

  Node deletion should follow strict lifecycle policies.

- How to Avoid:

  • Use node cordon + drain before deletion.

  • Monitor CSI cleanup completion before proceeding.

Scenario #365: Retained PV Blocks New Claim Binding with Identical Name

- Category: Storage

- Environment: Kubernetes v1.21, NFS

- Summary: A PV stuck in Released state with Retain policy blocked new PVCs from binding with the same name.

- What Happened:Deleted old PVC and recreated a new one with the same name, but it stayed Pending.

- Diagnosis Steps:

  • PV was in Released, PVC was Pending.

  • Events: PVC is not bound.

  Root Cause: Retained PV still owned the identity, blocking rebinding.

- Fix/Workaround:

  • Manually deleted the old PV to allow dynamic provisioning.

- Lessons Learned:

  Retain policies require admin cleanup.

- How to Avoid:

  • Use Delete policy for short-lived PVCs.

  • Automate orphan PV audits.

Scenario #366: CSI Plugin Panic on Missing Mount Option

- Category: Storage

- Environment: Kubernetes v1.26, custom CSI plugin

- Summary: Missing mountOptions in StorageClass led to runtime nil pointer exception in CSI driver.

- What Happened:StorageClass defined mountOptions: null, causing driver to crash during attach.

- Diagnosis Steps:

  • CSI logs showed panic: nil pointer dereference.

  • StorageClass YAML had an empty mountOptions: field.

  Root Cause: Plugin didn't check for nil before reading options.

- Fix/Workaround:

  • Removed mountOptions: from manifest.

  • Patched CSI driver to add nil checks.

- Lessons Learned:

  CSI drivers must gracefully handle incomplete specs.

- How to Avoid:

  • Validate StorageClass manifests.

  • Write defensive CSI plugin code.

Scenario #367: Pod Fails to Mount Volume Due to SELinux Context Mismatch

- Category: Storage

- Environment: Kubernetes v1.24, RHEL with SELinux enforcing

- Summary: Pod failed to mount volume due to denied SELinux permissions.

- What Happened:Volume was created with an incorrect SELinux context, preventing pod access.

- Diagnosis Steps:

  • Pod logs: permission denied.

  • dmesg showed SELinux AVC denial.

  Root Cause: Volume not labeled with container_file_t.

- Fix/Workaround:

  • Relabeled volume with chcon -Rt container_file_t /data.

- Lessons Learned:

  SELinux can silently block mounts.

- How to Avoid:

• Use CSI drivers that support SELinux integration.

• Validate volume contexts post-provisioning.

Scenario #368: VolumeExpansion on Bound PVC Fails Due to Pod Running

- Category: Storage

- Environment: Kubernetes v1.25, GCP PD

- Summary: PVC resize operation failed because the pod using it was still running.

- What Happened:Tried to resize a PVC while its pod was active.

- Diagnosis Steps:

  • PVC showed Resizing then back to Bound.

  • Events: PVC resize failed while volume in use.

  Root Cause: Filesystem resize required pod to restart.

- Fix/Workaround:

  • Deleted pod to trigger offline volume resize.

  • PVC then showed FileSystemResizePending → Bound.

- Lessons Learned:

Some resizes need pod restart.

- How to Avoid:

  • Plan PVC expansion during maintenance.
  • Use fsResizePolicy: "OnRestart" if supported.

Scenario #369: CSI Driver Memory Leak on Volume Detach Loop

- Category: Storage

- Environment: Kubernetes v1.24, external CSI

- Summary: CSI plugin leaked memory due to improper garbage collection on detach failure loop.

- What Happened:Detach failed repeatedly due to stale metadata, causing plugin to grow in memory use.

- Diagnosis Steps:

  • Plugin memory exceeded 1GB.
  • Logs showed repeated detach failed with no backoff.

  Root Cause: Driver retry loop without cleanup or GC.

- Fix/Workaround:

• Restarted CSI plugin.

• Patched driver to implement exponential backoff.

- Lessons Learned:

CSI error paths need memory safety.

- How to Avoid:

• Stress-test CSI paths for failure.

• Add Prometheus memory alerts for plugins.

Scenario #370: Volume Mount Timeout Due to Slow Cloud API

- Category: Storage

- Environment: Kubernetes v1.23, Azure Disk CSI

- Summary: During a cloud outage, Azure Disk operations timed out, blocking pod mounts.

- What Happened:Pods remained in ContainerCreating due to delayed volume attachment.

- Diagnosis Steps:

• Event logs: timed out waiting for attach.

• Azure portal showed degraded disk API service.

Root Cause: Cloud provider API latency blocked CSI attach.

- Fix/Workaround:

  • Waited for Azure API to stabilize.
  • Used local PVs for critical workloads moving forward.

- Lessons Learned:

  Cloud API reliability is a hidden dependency.

- How to Avoid:

  • Use local volumes or ephemeral storage for high-availability needs.
  • Monitor CSI attach/detach durations.

Scenario #371: Volume Snapshot Restore Misses Application Consistency

- Category: Storage

- Environment: Kubernetes v1.26, Velero with CSI VolumeSnapshot

- Summary: Snapshot restore completed successfully, but restored app data was corrupt.

- What Happened:A volume snapshot was taken while the database was mid-write. Restore completed, but database wouldn't start due to file inconsistencies.

- Diagnosis Steps:

  • Restored volume had missing WAL files.

  • Database logs showed corruption errors.

  • Snapshot logs showed no pre-freeze hook execution.

  Root Cause: No coordination between snapshot and application quiescence.

- Fix/Workaround:

  • Integrated pre-freeze and post-thaw hooks via Velero Restic.

  • Enabled application-aware backups.

- Lessons Learned:

  Volume snapshot ≠ app-consistent backup.

- How to Avoid:

  • Use app-specific backup tools or hooks.

  • Never snapshot during heavy write activity.

Scenario #372: File Locking Issue Between Multiple Pods on NFS

- Category: Storage

- Environment: Kubernetes v1.22, NFS with ReadWriteMany

- Summary: Two pods wrote to the same file concurrently, causing lock conflicts and data loss.

- What Happened:Lack of advisory file locking on the NFS server led to race conditions between pods.

- Diagnosis Steps:

  • Log files had overlapping, corrupted data.

  • File locks were not honored.

  Root Cause: POSIX locks not enforced reliably over NFS.

- Fix/Workaround:

  • Introduced flock-based locking in application code.

  • Used local persistent volume instead for critical data.

- Lessons Learned:

  NFS doesn’t guarantee strong file locking semantics.

- How to Avoid:

  • Architect apps to handle distributed file access carefully.

  • Avoid shared writable files unless absolutely needed.

Scenario #373: Pod Reboots Erase Data on EmptyDir Volume

- Category: Storage

- Environment: Kubernetes v1.24, default EmptyDir

- Summary: Pod restarts caused in-memory volume to be wiped, resulting in lost logs.

- What Happened:Logging container used EmptyDir with memory medium. Node rebooted, and logs were lost.

- Diagnosis Steps:

  • Post-reboot, EmptyDir was reinitialized.

  • Logs had disappeared from the container volume.

  Root Cause: EmptyDir with medium: Memory is ephemeral and tied to node lifecycle.

- Fix/Workaround:

  • Switched to hostPath for logs or persisted to object storage.

- Lessons Learned:

  Understand EmptyDir behavior before using for critical data.

- How to Avoid:

  • Use PVs or centralized logging for durability.

  • Avoid medium: Memory unless necessary.

Scenario #374: PVC Resize Fails on In-Use Block Device

- Category: Storage

- Environment: Kubernetes v1.25, CSI with block mode

- Summary: PVC expansion failed for a block device while pod was still running.

- What Happened:Attempted to resize a raw block volume without terminating the consuming pod.

- Diagnosis Steps:

  • PVC stuck in Resizing.
  • Logs: device busy.

  Root Cause: Some storage providers require offline resizing for block devices.

- Fix/Workaround:

  • Stopped the pod and retried resize.

- Lessons Learned:

  Raw block volumes behave differently than filesystem PVCs.

- How to Avoid:

  • Schedule maintenance windows for volume changes.

• Know volume mode differences.

Scenario #375: Default StorageClass Prevents PVC Binding to Custom Class

- Category: Storage

- Environment: Kubernetes v1.23, GKE

- Summary: A PVC remained in Pending because the default StorageClass kept getting assigned instead of a custom one.

- What Happened:PVC YAML didn’t specify storageClassName, so the default one was used.

- Diagnosis Steps:

• PVC described with wrong StorageClass.

• Events: no matching PV.

Root Cause: Default StorageClass mismatch with intended PV type.

- Fix/Workaround:

• Explicitly set storageClassName in the PVC.

- Lessons Learned:

Implicit defaults can cause hidden behavior.

- How to Avoid:

  • Always specify StorageClass explicitly in manifests.
  • Audit your cluster's default classes.

Scenario #376: Ceph RBD Volume Mount Failure Due to Kernel Mismatch

- Category: Storage

- Environment: Kubernetes v1.21, Rook-Ceph

- Summary: Mounting Ceph RBD volume failed after a node kernel upgrade.

- What Happened:The new kernel lacked required RBD modules.

- Diagnosis Steps:

  • dmesg showed rbd: module not found.
  • CSI logs indicated mount failed.

  Root Cause: Kernel modules not pre-installed after OS patching.

- Fix/Workaround:

  • Reinstalled kernel modules and rebooted node.

- Lessons Learned:

Kernel upgrades can silently break storage drivers.

- How to Avoid:

• Validate CSI compatibility post-upgrade.

• Use DaemonSet to check required modules.

Scenario #377: CSI Volume Cleanup Delay Leaves Orphaned Devices

- Category: Storage

- Environment: Kubernetes v1.24, Azure Disk CSI

- Summary: Volume deletion left orphaned devices on the node, consuming disk space.

- What Happened:Node failed to clean up mount paths after volume detach due to a kubelet bug.

- Diagnosis Steps:

• Found stale device mounts in /var/lib/kubelet/plugins/kubernetes.io/csi.

Root Cause: Kubelet failed to unmount due to corrupted symlink.

- Fix/Workaround:

• Manually removed symlinks and restarted kubelet.

- Lessons Learned:

  CSI volume cleanup isn’t always reliable.

- How to Avoid:

  • Monitor stale mounts.

  • Automate cleanup scripts in node maintenance routines.

Scenario #378: Immutable ConfigMap Used in CSI Sidecar Volume Mount

- Category: Storage

- Environment: Kubernetes v1.23, EKS

- Summary: CSI sidecar depended on a ConfigMap that was updated, but volume behavior didn’t change.

- What Happened:Sidecar didn’t restart, so old config was retained.

- Diagnosis Steps:

  • Volume behavior didn't reflect updated parameters.

  • Verified sidecar was still running with old config.

  Root Cause: ConfigMap change wasn’t detected because it was mounted as a volume.

- Fix/Workaround:

  • Restarted CSI sidecar pods.

- Lessons Learned:
  Mounting ConfigMaps doesn't auto-reload them.

- How to Avoid:

  • Use checksum/config annotations to force rollout.
  • Don't rely on in-place ConfigMap mutation.

Scenario #379: PodMount Denied Due to SecurityContext Constraints

- Category: Storage

- Environment: Kubernetes v1.25, OpenShift with SCCs

- Summary: Pod failed to mount PVC due to restricted SELinux type in pod's security context.

- What Happened:OpenShift SCC prevented the pod from mounting a volume with a mismatched SELinux context.

- Diagnosis Steps:

  • Events: permission denied during mount.

• Reviewed SCC and found allowedSELinuxOptions was too strict.

Root Cause: Security policies blocked mount operation.

- Fix/Workaround:

• Modified SCC to allow required context or used correct volume labeling.

- Lessons Learned:

Storage + security integration is often overlooked.

- How to Avoid:

• In tightly controlled environments, align volume labels with pod policies.

• Audit SCCs with volume access in mind.

Scenario #380: VolumeProvisioner Race Condition Leads to Duplicated PVC

- Category: Storage

- Environment: Kubernetes v1.24, CSI with dynamic provisioning

- Summary: Simultaneous provisioning requests created duplicate PVs for a single PVC.

- What Happened:PVC provisioning logic retried rapidly, and CSI provisioner created two volumes.

- Diagnosis Steps:

  • Observed two PVs with same claimRef.

  • Events showed duplicate provision succeeded entries.

  Root Cause: CSI controller did not lock claim state.

- Fix/Workaround:

  • Patched CSI controller to implement idempotent provisioning.

- Lessons Learned:

  CSI must be fault-tolerant to API retries.

- How to Avoid:

  • Ensure CSI drivers enforce claim uniqueness.

  • Use exponential backoff and idempotent logic.

Scenario #381: PVC Bound to Deleted PV After Restore

- Category: Storage

- Environment: Kubernetes v1.25, Velero restore with CSI driver

- Summary: Restored PVC bound to a PV that no longer existed, causing stuck pods.

- What Happened:During a cluster restore, PVC definitions were restored before their associated PVs. The missing PV names were still referenced.

- Diagnosis Steps:

  • PVCs stuck in Pending state.

  • Events: PV does not exist.

  • Velero logs showed PVCs restored first.

  Root Cause: Restore ordering issue in backup tool.

- Fix/Workaround:

  • Deleted and re-created PVCs manually or re-triggered restore in correct order.

- Lessons Learned:

  PVC-PV binding is tightly coupled.

- How to Avoid:

  • Use volume snapshot restores or ensure PVs are restored before PVCs.

  • Validate backup tool restore ordering.

Scenario #382: Unexpected Volume Type Defaults to HDD Instead of SSD

- Category: Storage

- Environment: Kubernetes v1.24, GKE with dynamic provisioning

- Summary: Volumes defaulted to HDD even though workloads needed SSD.

- What Happened:StorageClass used default pd-standard instead of pd-ssd.

- Diagnosis Steps:

  • IOPS metrics showed high latency.

  • Checked StorageClass: wrong type.

  Root Cause: Implicit default used in dynamic provisioning.

- Fix/Workaround:

  • Updated manifests to explicitly reference pd-ssd.

- Lessons Learned:

  Defaults may not match workload expectations.

- How to Avoid:

  • Always define storage class with performance explicitly.

  • Audit default class across environments.

Scenario #383: ReclaimPolicy Retain Caused Resource Leaks

- Category: Storage

- Environment: Kubernetes v1.22, bare-metal CSI

- Summary: Deleting PVCs left behind unused PVs and disks.

- What Happened:PVs had ReclaimPolicy: Retain, so disks weren't deleted.

- Diagnosis Steps:

  • PVs stuck in Released state.
  • Disk usage on nodes kept increasing.

  Root Cause: Misconfigured reclaim policy.

- Fix/Workaround:

  • Manually cleaned up PVs and external disk artifacts.

- Lessons Learned:

  Retain policy requires manual lifecycle management.

- How to Avoid:

  • Use Delete for ephemeral workloads.
  • Periodically audit released PVs.

Scenario #384: ReadWriteOnce PVC Mounted by Multiple Pods

- Category: Storage

- Environment: Kubernetes v1.23, AWS EBS

- Summary: Attempt to mount a ReadWriteOnce PVC on two pods in different AZs failed silently.

- What Happened:Pods scheduled across AZs; EBS volume couldn't attach to multiple nodes.

- Diagnosis Steps:

  • Pods stuck in ContainerCreating.

  • Events showed volume not attachable.

  Root Cause: ReadWriteOnce restriction and AZ mismatch.

- Fix/Workaround:

  • Updated deployment to use ReadWriteMany (EFS) for shared access.

- Lessons Learned:

  RWX vs RWO behavior varies by volume type.

- How to Avoid:

  • Use appropriate access modes per workload.

  • Restrict scheduling to compatible zones.

Scenario #385: VolumeAttach Race on StatefulSet Rolling Update

- Category: Storage

- Environment: Kubernetes v1.26, StatefulSet with CSI driver

- Summary: Volume attach operations failed during parallel pod updates.

- What Happened:Two pods in a StatefulSet update attempted to use the same PVC briefly due to quick scale down/up.

- Diagnosis Steps:

  • Events: Multi-Attach error for volume.
  • CSI logs showed repeated attach/detach.

  Root Cause: StatefulSet update policy did not wait for volume detachment.

- Fix/Workaround:

  • Set podManagementPolicy: OrderedReady.

- Lessons Learned:

  StatefulSet updates need to be serialized with volume awareness.

- How to Avoid:

• Tune StatefulSet rollout policies.

• Monitor CSI attach/detach metrics.

Scenario #386: CSI Driver CrashLoop Due to Missing Node Labels

- Category: Storage

- Environment: Kubernetes v1.24, OpenEBS CSI

- Summary: CSI sidecars failed to initialize due to missing node topology labels.

- What Happened:A node upgrade wiped custom labels needed for topology-aware provisioning.

- Diagnosis Steps:

• Logs: missing topology key node label.

• CSI pods in CrashLoopBackOff.

Root Cause: Topology-based provisioning misconfigured.

- Fix/Workaround:

• Reapplied node labels and restarted sidecars.

- Lessons Learned:

Custom node labels are critical for CSI topology hints.

- How to Avoid:

  • Enforce node label consistency using DaemonSets or node admission webhooks.

Scenario #387: PVC Deleted While Volume Still Mounted

- Category: Storage

- Environment: Kubernetes v1.22, on-prem CSI

- Summary: PVC deletion didn't unmount volume due to finalizer stuck on pod.

- What Happened:Pod was terminating but stuck, so volume detach never happened.

- Diagnosis Steps:

  • PVC deleted, but disk remained attached.
  • Pod in Terminating state for hours.

  Root Cause: Finalizer logic bug in kubelet.

- Fix/Workaround:

  • Force deleted pod, manually detached volume.

- Lessons Learned:

  Volume lifecycle is tied to pod finalization.

- How to Avoid:

  • Monitor long-running Terminating pods.

  • Use proper finalizer cleanup logic.

Scenario #388: In-Tree Volume Plugin Migration Caused Downtime

- Category: Storage

- Environment: Kubernetes v1.25, GKE

- Summary: GCE PD plugin migration to CSI caused volume mount errors.

- What Happened:After upgrade, in-tree plugin was disabled but CSI driver wasn’t fully configured.

- Diagnosis Steps:

  • Events: failed to provision volume.

  • CSI driver not installed.

  Root Cause: Incomplete migration preparation.

- Fix/Workaround:

• Re-enabled legacy plugin until CSI was functional.

- Lessons Learned:

Plugin migration is not automatic.

- How to Avoid:

• Review CSI migration readiness for your storage before upgrades.

Scenario #389: Overprovisioned Thin Volumes Hit Underlying Limit

- Category: Storage

- Environment: Kubernetes v1.24, LVM-based CSI

- Summary: Thin-provisioned volumes ran out of physical space, affecting all pods.

- What Happened:Overcommitted volumes filled up the disk pool.

- Diagnosis Steps:

• df on host showed 100% disk.

• LVM pool full, volumes became read-only.

Root Cause: No enforcement of provisioning limits.

- Fix/Workaround:

  • Resized physical disk and added monitoring.

- Lessons Learned:

  Thin provisioning must be paired with storage usage enforcement.

- How to Avoid:

  • Monitor volume pool usage.

  • Set quotas or alerts for overcommit.

Scenario #390: Dynamic Provisioning Failure Due to Quota Exhaustion

- Category: Storage

- Environment: Kubernetes v1.26, vSphere CSI

- Summary: PVCs failed to provision silently due to exhausted storage quota.

- What Happened:Storage backend rejected volume create requests.

- Diagnosis Steps:

  • PVC stuck in Pending.

  • CSI logs: quota exceeded.

Root Cause: Backend quota exceeded without Kubernetes alerting.

- Fix/Workaround:

• Increased quota or deleted old volumes.

- Lessons Learned:

Kubernetes doesn’t surface backend quota status clearly.

- How to Avoid:

• Integrate storage backend alerts into cluster monitoring.

• Tag and age out unused PVCs periodically.

Scenario #391: PVC Resizing Didn’t Expand Filesystem Automatically

- Category: Storage

- Environment: Kubernetes v1.24, AWS EBS, ext4 filesystem

- Summary: PVC was resized but the pod’s filesystem didn’t reflect the new size.

- What Happened:The PersistentVolume was expanded, but the pod using it didn’t see the increased size until restarted.

- Diagnosis Steps:

• df -h inside the pod showed old capacity.

• PVC showed updated size in Kubernetes.

Root Cause: Filesystem expansion requires a pod restart unless using CSI drivers with ExpandInUse support.

- Fix/Workaround:

• Restarted the pod to trigger filesystem expansion.

- Lessons Learned:

Volume expansion is two-step: PV resize and filesystem resize.

- How to Avoid:

• Use CSI drivers that support in-use expansion.

• Add automation to restart pods after volume resize.

Scenario #392: StatefulSet Pods Lost Volume Data After Node Reboot

- Category: Storage

- Environment: Kubernetes v1.22, local-path-provisioner

- Summary: Node reboots caused StatefulSet volumes to disappear due to ephemeral local storage.

- What Happened:After node maintenance, pods were rescheduled and couldn’t find their PVC data.

- Diagnosis Steps:

  • ls inside pod showed empty volumes.
  • PVCs bound to node-specific paths that no longer existed.

  Root Cause: Using local-path provisioner without persistence guarantees.

- Fix/Workaround:

  • Migrated to network-attached persistent storage (NFS/CSI).

- Lessons Learned:

  Local storage is node-specific and non-resilient.

- How to Avoid:

  • Use proper CSI drivers with data replication for StatefulSets.

Scenario #393: VolumeSnapshots Failed to Restore with Immutable Fields

- Category: Storage

- Environment: Kubernetes v1.25, VolumeSnapshot API

- Summary: Restore operation failed due to immutable PVC spec fields like access mode.

- What Happened:Attempted to restore snapshot into a PVC with modified parameters.

- Diagnosis Steps:

  • Error: cannot change accessMode after creation.

  Root Cause: Snapshot restore tried to override immutable PVC fields.

- Fix/Workaround:

  • Created a new PVC with correct parameters and attached manually.

- Lessons Learned:

  PVC fields are not override-safe during snapshot restores.

- How to Avoid:

  • Restore into newly created PVCs.

  • Match snapshot PVC spec exactly.

Scenario #394: GKE Autopilot PVCs Stuck Due to Resource Class Conflict

- Category: Storage

- Environment: GKE Autopilot, dynamic PVC provisioning

- Summary: PVCs remained in Pending state due to missing resource class binding.

- What Happened:GKE Autopilot required both PVC and pod to define compatible resourceClassName.

- Diagnosis Steps:

  • Events: No matching ResourceClass.

  • Pod log: PVC resource class mismatch.

  Root Cause: Autopilot restrictions on dynamic provisioning.

- Fix/Workaround:

  • Updated PVCs and workload definitions to specify supported resource classes.

- Lessons Learned:

  GKE Autopilot enforces stricter policies on storage.

- How to Avoid:

  • Follow GKE Autopilot documentation carefully.

  • Avoid implicit defaults in manifests.

Scenario #395: Cross-Zone Volume Scheduling Failed in Regional Cluster

- Category: Storage

- Environment: Kubernetes v1.24, GKE regional cluster

- Summary: Pods failed to schedule because volumes were provisioned in a different zone than the node.

- What Happened:Regional cluster scheduling pods to one zone while PVCs were created in another.

- Diagnosis Steps:

  • Events: FailedScheduling: volume not attachable.

  Root Cause: Storage class used zonal disks instead of regional.

- Fix/Workaround:

  • Updated storage class to use regional persistent disks.

- Lessons Learned:

  Volume zone affinity must match cluster layout.

- How to Avoid:

  • Use regional disks in regional clusters.

  • Always define zone spreading policy explicitly.

Scenario #396: Stuck Finalizers on Deleted PVCs Blocking Namespace Deletion

- Category: Storage

- Environment: Kubernetes v1.22, CSI driver

- Summary: Finalizers on PVCs blocked namespace deletion for hours.

- What Happened:Namespace was stuck in Terminating due to PVCs with finalizers not being properly removed.

- Diagnosis Steps:

  • Checked PVC YAML: finalizers section present.

  • Logs: CSI controller error during cleanup.

  Root Cause: CSI cleanup failed due to stale volume handles.

- Fix/Workaround:

  • Patched PVCs to remove finalizers manually.

- Lessons Learned:

  Finalizers can hang namespace deletion.

- How to Avoid:

  • Monitor PVCs with stuck finalizers.

• Regularly validate volume plugin cleanup.

Scenario #397: CSI Driver Upgrade Corrupted Volume Attachments

- Category: Storage

- Environment: Kubernetes v1.23, OpenEBS

- Summary: CSI driver upgrade introduced a regression causing volume mounts to fail.

- What Happened:After a helm-based CSI upgrade, pods couldn’t mount volumes.

- Diagnosis Steps:

  • Logs: mount timeout errors.
  • CSI logs showed broken symlinks.

  Root Cause: Helm upgrade deleted old CSI socket paths before new one started.

- Fix/Workaround:

  • Rolled back to previous CSI driver version.

- Lessons Learned:

  Upgrades should always be tested in staging clusters.

- How to Avoid:

• Perform canary upgrades.

• Backup CSI configurations and verify volume health post-upgrade.

Scenario #398: Stale Volume Handles After Disaster Recovery Cutover

- Category: Storage

- Environment: Kubernetes v1.25, Velero restore to DR cluster

- Summary: Stale volume handles caused new PVCs to fail provisioning.

- What Happened:Restored PVs referenced non-existent volume handles in new cloud region.

- Diagnosis Steps:

• CSI logs: volume handle not found.

• kubectl describe pvc: stuck in Pending.

Root Cause: Velero restore didn’t remap volume handles for the DR environment.

- Fix/Workaround:

• Manually edited PV specs or recreated PVCs from scratch.

- Lessons Learned:

Volume handles are environment-specific.

- How to Avoid:

  • Customize Velero restore templates.
  • Use snapshots or backups that are region-agnostic.

Scenario #399: Application Wrote Outside Mounted Path and Lost Data

- Category: Storage

- Environment: Kubernetes v1.24, default mountPath

- Summary: Application wrote logs to /tmp, not mounted volume, causing data loss on pod eviction.

- What Happened:Application configuration didn’t match the PVC mount path.

- Diagnosis Steps:

  • Pod deleted → logs disappeared.
  • PVC had no data.

  Root Cause: Application not configured to use the mounted volume path.

- Fix/Workaround:

• Updated application config to write into the mount path.

- Lessons Learned:

Mounted volumes don’t capture all file writes by default.

- How to Avoid:

• Review app config during volume integration.

• Validate mount paths with a test write-read cycle.

Scenario #400: Cluster Autoscaler Deleted Nodes with Mounted Volumes

- Category: Storage

- Environment: Kubernetes v1.23, AWS EKS with CA

- Summary: Cluster Autoscaler aggressively removed nodes with attached volumes, causing workload restarts.

- What Happened:Nodes were deemed underutilized and deleted while volumes were still mounted.

- Diagnosis Steps:

• Volumes detached mid-write, causing file corruption.

• Events showed node scale-down triggered by CA.

Root Cause: No volume-aware protection in CA.

- Fix/Workaround:

  • Enabled --balance-similar-node-groups and --skip-nodes-with-local-storage.

- Lessons Learned:
  Cluster Autoscaler must be volume-aware.

- How to Avoid:

  • Configure CA to respect mounted volumes.
  • Tag volume-critical nodes as unschedulable before scale-down.

Category 5: Scaling & Load

Scenario #401: HPA Didn't Scale Due to Missing Metrics Server

- Category: Scaling & Load

- Environment: Kubernetes v1.22, Minikube

- Summary: Horizontal Pod Autoscaler (HPA) didn’t scale pods as expected.

- What Happened:HPA showed unknown metrics and pod count remained constant despite CPU stress.

- Diagnosis Steps:

  • kubectl get hpa showed Metrics not available.

  • Confirmed metrics-server not installed.

  Root Cause: Metrics server was missing, which is required by HPA for decision making.

- Fix/Workaround:

  • Installed metrics-server using official manifests.

- Lessons Learned:

  HPA silently fails without metrics-server.

- How to Avoid:

  • Include metrics-server in base cluster setup.

  • Monitor HPA status regularly.

Scenario #402: CPU Throttling Prevented Effective Autoscaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, EKS, Burstable QoS

- Summary: Application CPU throttled even under low usage, leading to delayed scaling.

- What Happened:HPA didn’t trigger scale-up due to misleading low CPU usage stats.

- Diagnosis Steps:

  • Metrics showed low CPU, but app performance was poor.

  • kubectl top pod confirmed low utilization.

  • cgroups showed heavy throttling.

  Root Cause: CPU limits were set too close to requests, causing throttling.

- Fix/Workaround:

  • Increased CPU limits or removed them entirely for key services.

- Lessons Learned:

  CPU throttling can suppress scaling metrics.

- How to Avoid:

  • Monitor cgroup throttling stats.

  • Tune CPU requests/limits carefully.

Scenario #403: Overprovisioned Pods Starved the Cluster

- Category: Scaling & Load

- Environment: Kubernetes v1.21, on-prem

- Summary: Aggressively overprovisioned pod resources led to failed scheduling and throttling.

- What Happened:Apps were deployed with excessive CPU/memory, blocking HPA and new workloads.

- Diagnosis Steps:

  • kubectl describe node: Insufficient CPU errors.

  • Top nodes showed 50% actual usage, 100% requested.

  Root Cause: Reserved resources were never used but blocked the scheduler.

- Fix/Workaround:

  • Adjusted requests/limits based on real usage.

- Lessons Learned:

  Resource requests ≠ real consumption.

- How to Avoid:

  • Right-size pods using VPA recommendations or Prometheus usage data.

Scenario #404: HPA and VPA Conflicted, Causing Flapping

- Category: Scaling & Load

- Environment: Kubernetes v1.25, GKE

- Summary: HPA scaled replicas based on CPU while VPA changed pod resources dynamically, creating instability.

- What Happened:HPA scaled up, VPA shrank resources → load spike → HPA scaled again.

- Diagnosis Steps:

  • Logs showed frequent pod terminations and creations.

  • Pod count flapped repeatedly.

  Root Cause: HPA and VPA were configured on the same deployment without proper coordination.

- Fix/Workaround:

  • Disabled VPA on workloads using HPA.

- Lessons Learned:

  HPA and VPA should be used carefully together.

- How to Avoid:

  • Use HPA for scale-out and VPA for fixed-size workloads.

  • Avoid combining on the same object.

Scenario #405: Cluster Autoscaler Didn't Scale Due to Pod Affinity Rules

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Workloads couldn't be scheduled and CA didn’t scale nodes because affinity rules restricted placement.

- What Happened:Pods failed to schedule and were stuck in Pending, but no scale-out occurred.

- Diagnosis Steps:

  • Events: FailedScheduling with affinity violations.

  • CA logs: “no matching node group”.

  Root Cause: Pod anti-affinity restricted nodes that CA could provision.

- Fix/Workaround:

  • Relaxed anti-affinity or labeled node groups appropriately.

- Lessons Learned:

  Affinity rules affect autoscaler decisions.

- How to Avoid:

  • Use soft affinity (preferredDuringScheduling) where possible.

• Monitor unschedulable pods with alerting.

Scenario #406: Load Test Crashed Cluster Due to Insufficient Node Quotas

- Category: Scaling & Load

- Environment: Kubernetes v1.24, AKS

- Summary: Stress test resulted in API server crash due to unthrottled pod burst.

- What Happened:Locust load test created hundreds of pods, exceeding node count limits.

- Diagnosis Steps:

  • API server latency spiked, etcd logs flooded.
  • Cluster hit node quota limit on Azure.

  Root Cause: No upper limit on replica count during load test; hit cloud provider limits.

- Fix/Workaround:

  • Added maxReplicas to HPA.
  • Throttled CI tests.

- Lessons Learned:

  CI/CD and load tests should obey cluster quotas.

- How to Avoid:

  • Monitor node count vs quota in metrics.
  • Set maxReplicas in HPA and cap CI workloads.

Scenario #407: Scale-To-Zero Caused Cold Starts and SLA Violations

- Category: Scaling & Load

- Environment: Kubernetes v1.25, KEDA + Knative

- Summary: Pods scaled to zero, but requests during cold start breached SLA.

- What Happened:First request after inactivity hit cold-start delay of ~15s.

- Diagnosis Steps:

  • Prometheus response latency showed spikes after idle periods.
  • Knative logs: cold-start events.

  Root Cause: Cold starts on scale-from-zero under high latency constraint.

- Fix/Workaround:

  • Added minReplicaCount: 1 to high-SLA services.

- Lessons Learned:

  Scale-to-zero saves cost, but not for latency-sensitive apps.

- How to Avoid:

  • Use minReplicaCount and warmers for performance-critical services.

Scenario #408: Misconfigured Readiness Probe Blocked HPA Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, DigitalOcean

- Summary: HPA didn't scale pods because readiness probes failed and metrics were not reported.

- What Happened:Misconfigured probe returned 404, making pods invisible to HPA.

- Diagnosis Steps:

  • kubectl describe pod: readiness failed.

  • kubectl get hpa: no metrics available.

  Root Cause: Failed readiness probes excluded pods from metrics aggregation.

- Fix/Workaround:

• Corrected readiness endpoint in manifest.

- Lessons Learned:

HPA only sees "ready" pods.

- How to Avoid:

• Validate probe paths before production.

• Monitor readiness failures via alerts.

Scenario #409: Custom Metrics Adapter Crashed, Breaking Custom HPA

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Prometheus Adapter

- Summary: Custom HPA didn’t function after metrics adapter pod crashed silently.

- What Happened:HPA relying on Prometheus metrics didn't scale for hours.

- Diagnosis Steps:

• kubectl get hpa: metric unavailable.

• Checked prometheus-adapter logs: crashloop backoff.

Root Cause: Misconfigured rules in adapter config caused panic.

- Fix/Workaround:

  • Fixed Prometheus query in adapter configmap.

- Lessons Learned:

  Custom HPA is fragile to adapter errors.

- How to Avoid:

  • Set alerts on prometheus-adapter health.

  • Validate custom queries before deploy.

Scenario #410: Application Didn't Handle Scale-In Gracefully

- Category: Scaling & Load

- Environment: Kubernetes v1.22, Azure AKS

- Summary: App lost in-flight requests during scale-down, causing 5xx spikes.

- What Happened:Pods were terminated abruptly during autoscaling down, mid-request.

- Diagnosis Steps:

  • Observed 502/504 errors in logs during scale-in events.

• No termination hooks present.

Root Cause: No preStop hooks or graceful shutdown handling in the app.

- Fix/Workaround:

• Implemented preStop hook with delay.

• Added graceful shutdown in app logic.

- Lessons Learned:

Scale-in should be as graceful as scale-out.

- How to Avoid:

• Always include termination handling in apps.

• Use terminationGracePeriodSeconds wisely.

Scenario #411: Cluster Autoscaler Ignored Pod PriorityClasses

- Category: Scaling & Load

- Environment: Kubernetes v1.25, AWS EKS with PriorityClasses

- Summary: Low-priority workloads blocked scaling of high-priority ones due to misconfigured Cluster Autoscaler.

- What Happened:High-priority pods remained pending, even though Cluster Autoscaler was active.

- Diagnosis Steps:

  • kubectl get pods --all-namespaces | grep Pending showed stuck critical workloads.

  • CA logs indicated scale-up denied due to resource reservation by lower-priority pods.

  Root Cause: Default CA config didn't preempt lower-priority pods.

- Fix/Workaround:

  • Enabled preemption.

  • Re-tuned PriorityClass definitions to align with business SLAs.

- Lessons Learned:

  CA doesn’t preempt unless explicitly configured.

- How to Avoid:

  • Validate PriorityClass behavior in test environments.

  • Use preemptionPolicy: PreemptLowerPriority for critical workloads.

Scenario #412: ReplicaSet Misalignment Led to Excessive Scale-Out

- Category: Scaling & Load

- Environment: Kubernetes v1.23, GKE

- Summary: A stale ReplicaSet with label mismatches caused duplicate pod scale-out.

- What Happened:Deployment scaled twice the required pod count after an upgrade.

- Diagnosis Steps:

  • kubectl get replicasets showed multiple active sets with overlapping match labels.
  • Pod count exceeded expected limits.

  Root Cause: A new deployment overlapped labels with an old one; HPA acted on both.

- Fix/Workaround:

  • Cleaned up old ReplicaSets.
  • Scoped matchLabels more tightly.

- Lessons Learned:

  Label discipline is essential for reliable scaling.

- How to Avoid:

  • Use distinct labels per version or release.
  • Automate cleanup of unused ReplicaSets.

Scenario #413: StatefulSet Didn't Scale Due to PodDisruptionBudget

- Category: Scaling & Load

- Environment: Kubernetes v1.26, AKS

- Summary: StatefulSet couldn’t scale-in during node pressure due to a restrictive PDB.

- What Happened:Nodes under memory pressure tried to evict pods, but eviction was blocked.

- Diagnosis Steps:

  • Checked kubectl describe pdb and kubectl get evictions.

  • Events showed "Cannot evict pod as it would violate PDB".

  Root Cause: PDB defined minAvailable: 100%, preventing any disruption.

- Fix/Workaround:

  • Adjusted PDB to tolerate one pod disruption.

- Lessons Learned:

  Aggressive PDBs block both scaling and upgrades.

- How to Avoid:

  • Use realistic minAvailable or maxUnavailable settings.

  • Review PDB behavior in test scaling operations.

Scenario #414: Horizontal Pod Autoscaler Triggered by Wrong Metric

- Category: Scaling & Load

- Environment: Kubernetes v1.24, DigitalOcean

- Summary: HPA used memory instead of CPU, causing unnecessary scale-ups.

- What Happened:Application scaled even under light CPU usage due to memory caching behavior.

- Diagnosis Steps:

  • HPA target: memory utilization.

  • kubectl top pods: memory always high due to in-memory cache.

  Root Cause: Application design led to consistently high memory usage.

- Fix/Workaround:

  • Switched HPA to CPU metric.

  • Tuned caching logic in application.

- Lessons Learned:

  Choose scaling metrics that reflect true load.

- How to Avoid:

  • Profile application behavior before configuring HPA.

• Avoid memory-based autoscaling unless necessary.

Scenario #415: Prometheus Scraper Bottlenecked Custom HPA Metrics

- Category: Scaling & Load

- Environment: Kubernetes v1.25, custom metrics + Prometheus Adapter

- Summary: Delays in Prometheus scraping caused lag in HPA reactions.

- What Happened:HPA lagged 1–2 minutes behind actual load spike.

- Diagnosis Steps:

  • prometheus-adapter logs showed stale data timestamps.
  • HPA scale-up occurred after delay.

  Root Cause: Scrape interval was 60s, making HPA respond too slowly.

- Fix/Workaround:

  • Reduced scrape interval for critical metrics.

- Lessons Learned:

  Scrape intervals affect autoscaler agility.

- How to Avoid:

• Match Prometheus scrape intervals with HPA polling needs.

• Use rate() or avg_over_time() to smooth metrics.

Scenario #416: Kubernetes Downscaled During Rolling Update

- Category: Scaling & Load

- Environment: Kubernetes v1.23, on-prem

- Summary: Pods were prematurely scaled down during rolling deployment.

- What Happened:Rolling update caused a drop in available replicas, triggering autoscaler.

- Diagnosis Steps:

• Observed spike in 5xx errors during update.

• HPA decreased replica count despite live traffic.

Root Cause: Deployment strategy interfered with autoscaling logic.

- Fix/Workaround:

• Tuned maxUnavailable and minReadySeconds.

• Added load-based HPA stabilization window.

- Lessons Learned:

  HPA must be aligned with rolling deployment behavior.

- How to Avoid:

  • Use behavior.scaleDown.stabilizationWindowSeconds.

  • Monitor scaling decisions during rollouts.

Scenario #417: KEDA Failed to Scale on Kafka Lag Metric

- Category: Scaling & Load

- Environment: Kubernetes v1.26, KEDA + Kafka

- Summary: Consumers didn’t scale out despite Kafka topic lag.

- What Happened:High message lag persisted but consumer replicas remained at baseline.

- Diagnosis Steps:

  • kubectl get scaledobject showed no trigger activation.

  • Logs: authentication to Kafka metrics endpoint failed.

  Root Cause: Incorrect TLS cert in KEDA trigger config.

- Fix/Workaround:

• Updated Kafka trigger auth to use correct secret.

- Lessons Learned:

External metric sources require secure, stable access.

- How to Avoid:

• Validate all trigger auth and endpoints before production.

• Alert on trigger activation failures.

Scenario #418: Spike in Load Exceeded Pod Init Time

- Category: Scaling & Load

- Environment: Kubernetes v1.24, self-hosted

- Summary: Sudden burst of traffic overwhelmed services due to slow pod boot time.

- What Happened:HPA triggered scale-out, but pods took too long to start. Users got errors.

- Diagnosis Steps:

• Noticed gap between scale-out and readiness.

• Startup probes took 30s+ per pod.

Root Cause: App container had heavy init routines.

- Fix/Workaround:

• Optimized Docker image layers and moved setup to init containers.

- Lessons Learned:

Scale-out isn’t instant; pod readiness matters.

- How to Avoid:

• Track ReadySeconds vs ReplicaScale delay.

• Pre-pull images and optimize pod init time.

Scenario #419: Overuse of Liveness Probes Disrupted Load Balance

- Category: Scaling & Load

- Environment: Kubernetes v1.21, bare metal

- Summary: Misfiring liveness probes killed healthy pods during load test.

- What Happened:Sudden scale-out introduced new pods, which were killed due to false negatives on liveness probes.

- Diagnosis Steps:

• Pod logs showed probe failures under high CPU.

• Readiness was OK, liveness killed them anyway.

Root Cause: CPU starvation during load caused probe timeouts.

- Fix/Workaround:

• Increased probe timeoutSeconds and failureThreshold.

- Lessons Learned:

Under load, even health checks need headroom.

- How to Avoid:

• Separate readiness from liveness logic.

• Gracefully handle CPU-heavy workloads.

Scenario #420: Scale-In Happened Before Queue Was Drained

- Category: Scaling & Load

- Environment: Kubernetes v1.26, RabbitMQ + consumers

- Summary: Consumers scaled in while queue still had unprocessed messages.

- What Happened:Queue depth remained, but pods were terminated.

- Diagnosis Steps:

  • Observed message backlog after autoscaler scale-in.

  • Consumers had no shutdown hook to drain queue.

  Root Cause: Scale-in triggered without consumer workload cleanup.

- Fix/Workaround:

  • Added preStop hook to finish queue processing.

- Lessons Learned:

  Consumers must handle shutdown gracefully.

- How to Avoid:

  • Track message queues with KEDA or custom metrics.

  • Add drain() logic on signal trap in consumer code.

Scenario #421: Node Drain Race Condition During Scale Down

- Category: Scaling & Load

- Environment: Kubernetes v1.23, GKE

- Summary: Node drain raced with pod termination, causing pod loss.

- What Happened:Pods were terminated while the node was still draining, leading to data loss.

- Diagnosis Steps:

  • kubectl describe node showed multiple eviction races.

  • Pod logs showed abrupt termination without graceful shutdown.

  Root Cause: Scale-down process didn’t wait for node draining to complete fully.

- Fix/Workaround:

  • Adjusted terminationGracePeriodSeconds for pods.

  • Introduced node draining delay in scaling policy.

- Lessons Learned:

  Node draining should be synchronized with pod termination.

- How to Avoid:

  • Use PodDisruptionBudget to ensure safe scaling.

  • Implement pod graceful shutdown hooks.

Scenario #422: HPA Disabled Due to Missing Resource Requests

- Category: Scaling & Load

- Environment: Kubernetes v1.22, AWS EKS

- Summary: Horizontal Pod Autoscaler (HPA) failed to trigger because resource requests weren't set.

- What Happened:HPA couldn't scale pods up despite high traffic due to missing CPU/memory resource requests.

- Diagnosis Steps:

  • kubectl describe deployment revealed missing resources.requests.

  • Logs indicated HPA couldn't fetch metrics without resource requests.

  Root Cause: Missing resource request fields prevented HPA from making scaling decisions.

- Fix/Workaround:

  • Set proper resources.requests in the deployment YAML.

- Lessons Learned:

  Always define resource requests to enable autoscaling.

- How to Avoid:

  • Define resource requests/limits for every pod.

  • Enable autoscaling based on requests/limits.

Scenario #423: Unexpected Overprovisioning of Pods

- Category: Scaling & Load

- Environment: Kubernetes v1.24, DigitalOcean

- Summary: Unnecessary pod scaling due to misconfigured resource limits.

- What Happened:Pods scaled up unnecessarily due to excessively high resource limits.

- Diagnosis Steps:

  • HPA logs showed frequent scale-ups even during low load.

  • Resource limits were higher than actual usage.

  Root Cause: Overestimated resource limits in pod configuration.

- Fix/Workaround:

  • Reduced resource limits to more realistic values.

- Lessons Learned:

  Proper resource allocation helps prevent scaling inefficiencies.

- How to Avoid:

  • Monitor resource consumption patterns before setting limits.

  • Use Kubernetes resource usage metrics to adjust configurations.

Scenario #424: Autoscaler Failed During StatefulSet Upgrade

- Category: Scaling & Load

- Environment: Kubernetes v1.25, AKS

- Summary: Horizontal scaling issues occurred during rolling upgrade of StatefulSet.

- What Happened:StatefulSet failed to scale out during a rolling upgrade, causing delayed availability of new pods.

- Diagnosis Steps:

  • Observed kubectl get pods showing delayed stateful pod restarts.

  • HPA did not trigger due to stuck pod state.

  Root Cause: Rolling upgrade conflicted with autoscaler logic due to StatefulSet constraints.

- Fix/Workaround:

  • Adjusted StatefulSet rollingUpdate strategy.

  • Tuned autoscaler thresholds for more aggressive scaling.

- Lessons Learned:

  Ensure compatibility between scaling and StatefulSet updates.

- How to Avoid:

  • Test upgrade and scaling processes in staging environments.
  • Separate stateful workloads from stateless ones for scaling flexibility.

Scenario #425: Inadequate Load Distribution in a Multi-AZ Setup

- Category: Scaling & Load

- Environment: Kubernetes v1.27, AWS EKS

- Summary: Load balancing wasn't even across availability zones, leading to inefficient scaling.

- What Happened:More traffic hit one availability zone (AZ), causing scaling delays in the other AZs.

- Diagnosis Steps:

  • Analyzed kubectl describe svc and found skewed traffic distribution.
  • Observed insufficient pod presence in multiple AZs.

  Root Cause: The Kubernetes service didn't properly distribute traffic across AZs.

- Fix/Workaround:

  • Updated service to use topologySpreadConstraints for better AZ distribution.

- Lessons Learned:

  Multi-AZ distribution requires proper spread constraints for effective scaling.

- How to Avoid:

  • Use topologySpreadConstraints in services to ensure balanced load.

  • Review multi-AZ architecture for traffic efficiency.

Scenario #426: Downscale Too Aggressive During Traffic Dips

- Category: Scaling & Load

- Environment: Kubernetes v1.22, GCP

- Summary: Autoscaler scaled down too aggressively during short traffic dips, causing pod churn.

- What Happened:Traffic decreased briefly, triggering a scale-in, only for the traffic to spike again.

- Diagnosis Steps:

  • HPA scaled down to 0 replicas during a brief traffic lull.

  • Pod churn noticed after every scale-in event.

  Root Cause: Aggressive scaling behavior set too low a minReplicas threshold.

- Fix/Workaround:

• Set a minimum of 1 replica for critical workloads.

• Tuned scaling thresholds to avoid premature downscaling.

- Lessons Learned:

Aggressive scaling policies can cause instability in unpredictable workloads.

- How to Avoid:

• Use minReplicas for essential workloads.

• Implement stabilization windows for both scale-up and scale-down.

Scenario #427: Insufficient Scaling Under High Ingress Traffic

- Category: Scaling & Load

- Environment: Kubernetes v1.26, NGINX Ingress Controller

- Summary: Pod autoscaling didn't trigger in time to handle high ingress traffic.

- What Happened:Ingress traffic surged, but HPA didn't trigger additional pods in time.

- Diagnosis Steps:

• Checked HPA configuration and metrics, found that HPA was based on CPU usage, not ingress traffic.

Root Cause: Autoscaling metric didn't account for ingress load.

- Fix/Workaround:

• Implemented custom metrics for Ingress traffic.

• Configured HPA to scale based on traffic load.

- Lessons Learned:

Use the right scaling metric for your workload.

- How to Avoid:

• Set custom metrics like ingress traffic for autoscaling.

• Regularly adjust metrics as load patterns change.

Scenario #428: Nginx Ingress Controller Hit Rate Limit on External API

- Category: Scaling & Load

- Environment: Kubernetes v1.25, AWS EKS

- Summary: Rate limits were hit on an external API during traffic surge, affecting service scaling.

- What Happened:Nginx Ingress Controller was rate-limited by an external API during a traffic surge.

- Diagnosis Steps:

  • Traffic logs showed 429 status codes for external API calls.

  • Observed HPA not scaling fast enough to handle the increased API request load.

  Root Cause: External API rate limiting was not considered in scaling decisions.

- Fix/Workaround:

  • Added retry logic for external API requests.

  • Adjusted autoscaling to consider both internal load and external API delays.

- Lessons Learned:

  Scaling should consider both internal and external load.

- How to Avoid:

  • Implement circuit breakers and retries for external dependencies.

  • Use comprehensive metrics for autoscaling decisions.

Scenario #429: Resource Constraints on Node Impacted Pod Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, on-prem

- Summary: Pod scaling failed due to resource constraints on nodes during high load.

- What Happened:Autoscaler triggered, but nodes lacked available resources, preventing new pods from starting.

- Diagnosis Steps:

  • kubectl describe nodes showed resource exhaustion.

  • kubectl get pods confirmed that scaling requests were blocked.

  Root Cause: Nodes were running out of resources during scaling decisions.

- Fix/Workaround:

  • Added more nodes to the cluster.

  • Increased resource limits for node pools.

- Lessons Learned:

  Cluster resource provisioning must be aligned with scaling needs.

- How to Avoid:

  • Regularly monitor node resource usage.

  • Use cluster autoscaling to add nodes as needed.

Scenario #430: Memory Leak in Application Led to Excessive Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.23, Azure AKS

- Summary: A memory leak in the app led to unnecessary scaling, causing resource exhaustion.

- What Happened:Application memory usage grew uncontrollably, causing HPA to continuously scale the pods.

- Diagnosis Steps:

  • kubectl top pods showed continuously increasing memory usage.

  • HPA logs showed scaling occurred without sufficient load.

  Root Cause: Application bug causing memory leak was misinterpreted as load spike.

- Fix/Workaround:

  • Identified and fixed the memory leak in the application code.

  • Tuned autoscaling to more accurately measure actual load.

- Lessons Learned:

  Memory issues can trigger excessive scaling; proper monitoring is critical.

- How to Avoid:

  • Implement application-level memory monitoring.

  • Set proper HPA metrics to differentiate load from resource issues.

Scenario #431: Inconsistent Pod Scaling During Burst Traffic

- Category: Scaling & Load

- Environment: Kubernetes v1.24, AWS EKS

- Summary: Pod scaling inconsistently triggered during burst traffic spikes, causing service delays.

- What Happened:A traffic burst caused sporadic scaling events that didn't meet demand, leading to delayed responses.

- Diagnosis Steps:

  • Observed scaling logs that showed pod scaling lagged behind traffic spikes.

  • Metrics confirmed traffic surges weren't matched by scaling.

  Root Cause: Insufficient scaling thresholds and long stabilization windows for HPA.

- Fix/Workaround:

  • Adjusted HPA settings to lower the stabilization window and set appropriate scaling thresholds.

- Lessons Learned:

  HPA scaling settings should be tuned to handle burst traffic effectively.

- How to Avoid:

• Use lower stabilization windows for quicker scaling reactions.

• Monitor scaling efficiency during traffic bursts.

Scenario #432: Auto-Scaling Hit Limits with StatefulSet

- Category: Scaling & Load

- Environment: Kubernetes v1.22, GCP

- Summary: StatefulSet scaling hit limits due to pod affinity constraints.

- What Happened:Auto-scaling did not trigger correctly due to pod affinity constraints limiting scaling.

- Diagnosis Steps:

• Found pod affinity rules restricted the number of eligible nodes for scaling.

• Logs showed pod scheduling failure during scale-up attempts.

Root Cause: Tight affinity rules prevented pods from being scheduled to new nodes.

- Fix/Workaround:

• Adjusted pod affinity rules to allow scaling across more nodes.

- Lessons Learned:

Pod affinity must be balanced with scaling needs.

- How to Avoid:

  • Regularly review affinity and anti-affinity rules when using HPA.
  • Test autoscaling scenarios with varying node configurations.

Scenario #433: Cross-Cluster Autoscaling Failures

- Category: Scaling & Load

- Environment: Kubernetes v1.21, Azure AKS

- Summary: Autoscaling failed across clusters due to inconsistent resource availability between regions.

- What Happened:Horizontal scaling issues arose when pods scaled across regions, leading to resource exhaustion.

- Diagnosis Steps:

  • Checked cross-cluster communication and found uneven resource distribution.
  • Found that scaling was triggered in one region but failed to scale in others.

  Root Cause: Resource discrepancies across regions caused scaling failures.

- Fix/Workaround:

  • Adjusted resource allocation policies to account for cross-cluster scaling.

• Ensured consistent resource availability across regions.

- Lessons Learned:

Cross-region autoscaling requires careful resource management.

- How to Avoid:

• Regularly monitor resources across clusters.

• Use a global view for autoscaling decisions.

Scenario #434: Service Disruption During Auto-Scaling of StatefulSet

- Category: Scaling & Load

- Environment: Kubernetes v1.24, AWS EKS

- Summary: StatefulSet failed to scale properly during maintenance, causing service disruption.

- What Happened:StatefulSet pods failed to scale correctly during a rolling update due to scaling policies not considering pod states.

- Diagnosis Steps:

• Logs revealed pods were stuck in a Pending state during scale-up.

• StatefulSet's rollingUpdate strategy wasn’t optimal.

Root Cause: StatefulSet scaling wasn’t fully compatible with the default rolling update strategy.

- Fix/Workaround:

  • Tuning the rollingUpdate strategy allowed pods to scale without downtime.

- Lessons Learned:

  StatefulSets require special handling during scale-up or down.

- How to Avoid:

  • Test scaling strategies with StatefulSets to avoid disruption.

  • Use strategies suited for the application type.

Scenario #435: Unwanted Pod Scale-down During Quiet Periods

- Category: Scaling & Load

- Environment: Kubernetes v1.23, GKE

- Summary: Autoscaler scaled down too aggressively during periods of low traffic, leading to resource shortages during traffic bursts.

- What Happened:Autoscaler reduced pod count during a quiet period, but didn’t scale back up quickly enough when traffic surged.

- Diagnosis Steps:

  • Investigated autoscaler settings and found low scaleDown stabilization thresholds.

  • Observed that scaling adjustments were made too aggressively.

  Root Cause: Too-sensitive scale-down triggers and lack of delay in scale-down events.

- Fix/Workaround:

  • Increased scaleDown stabilization settings to prevent rapid pod removal.

  • Adjusted thresholds to delay scale-down actions.

- Lessons Learned:

  Autoscaler should be tuned for traffic fluctuations.

- How to Avoid:

  • Implement proper scale-up and scale-down stabilization windows.

  • Fine-tune autoscaling thresholds based on real traffic patterns.

Scenario #436: Cluster Autoscaler Inconsistencies with Node Pools

- Category: Scaling & Load

- Environment: Kubernetes v1.25, GCP

- Summary: Cluster Autoscaler failed to trigger due to node pool constraints.

- What Happened:Nodes were not scaled when needed because Cluster Autoscaler couldn't add resources due to predefined node pool limits.

- Diagnosis Steps:

  • Examined autoscaler logs, revealing node pool size limits were blocking node creation.

  • Cluster metrics confirmed high CPU usage but no new nodes were provisioned.

  Root Cause: Cluster Autoscaler misconfigured node pool limits.

- Fix/Workaround:

  • Increased node pool size limits to allow autoscaling.

  • Adjusted autoscaler settings to better handle resource spikes.

- Lessons Learned:

  Autoscaling requires proper configuration of node pools.

- How to Avoid:

  • Ensure that node pool limits are set high enough for scaling.

  • Monitor autoscaler logs to catch issues early.

Scenario #437: Disrupted Service During Pod Autoscaling in StatefulSet

- Category: Scaling & Load

- Environment: Kubernetes v1.22, AWS EKS

- Summary: Pod autoscaling in a StatefulSet led to disrupted service due to the stateful nature of the application.

- What Happened:Scaling actions impacted the stateful application, causing data integrity issues.

- Diagnosis Steps:

  • Reviewed StatefulSet logs and found missing data after scale-ups.

  • Found that scaling interfered with pod affinity, causing service disruption.

  Root Cause: StatefulSet’s inherent behavior combined with pod autoscaling led to resource conflicts.

- Fix/Workaround:

  • Disabled autoscaling for stateful pods and adjusted configuration for better handling of stateful workloads.

- Lessons Learned:

  StatefulSets need special consideration when scaling.

- How to Avoid:

  • Avoid autoscaling for stateful workloads unless fully tested and adjusted.

Scenario #438: Slow Pod Scaling During High Load

- Category: Scaling & Load

- Environment: Kubernetes v1.26, DigitalOcean

- Summary: Autoscaling pods didn’t trigger quickly enough during sudden high-load events, causing delays.

- What Happened:Scaling didn’t respond fast enough during high load, leading to poor user experience.

- Diagnosis Steps:

  • Analyzed HPA logs and metrics, which showed a delayed response to traffic spikes.

  • Monitored pod resource utilization which showed excess load.

  Root Cause: Scaling policy was too conservative with high-load thresholds.

- Fix/Workaround:

  • Adjusted HPA to trigger scaling at lower thresholds.

- Lessons Learned:

  Autoscaling policies should respond more swiftly under high-load conditions.

- How to Avoid:

  • Fine-tune scaling thresholds for different traffic patterns.

• Use fine-grained metrics to adjust scaling behavior.

Scenario #439: Autoscaler Skipped Scale-up Due to Incorrect Metric

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Autoscaler skipped scale-up because it was using the wrong metric for scaling.

- What Happened:HPA was using memory usage as the metric, but CPU usage was the actual bottleneck.

- Diagnosis Steps:

• HPA logs showed autoscaler ignored CPU metrics in favor of memory.

• Metrics confirmed high CPU usage and low memory.

Root Cause: HPA was configured to scale based on memory instead of CPU usage.

- Fix/Workaround:

• Reconfigured HPA to scale based on CPU metrics.

- Lessons Learned:

Choose the correct scaling metric for the workload.

- How to Avoid:

  • Periodically review scaling metric configurations.

  • Test scaling behaviors using multiple types of metrics.

Scenario #440: Scaling Inhibited Due to Pending Jobs in Queue

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Azure AKS

- Summary: Pod scaling was delayed because jobs in the queue were not processed fast enough.

- What Happened:A backlog of jobs created delays in scaling, as the job queue was overfilled.

- Diagnosis Steps:

  • Examined job logs, which confirmed long processing times for queued tasks.

  • Found that the HPA didn't account for the job queue backlog.

  Root Cause: Insufficient pod scaling in response to job queue size.

- Fix/Workaround:

  • Added job queue monitoring metrics to scaling triggers.

  • Adjusted HPA to trigger based on job queue size and pod workload.

- Lessons Learned:

  Scale based on queue and workload, not just traffic.

- How to Avoid:

  • Implement queue size-based scaling triggers.

  • Use advanced metrics for autoscaling decisions.

Scenario #441: Scaling Delayed Due to Incorrect Resource Requests

- Category: Scaling & Load

- Environment: Kubernetes v1.24, AWS EKS

- Summary: Pod scaling was delayed because of incorrectly set resource requests, leading to resource over-provisioning.

- What Happened:Pods were scaled up, but they failed to start due to overly high resource requests that exceeded available node capacity.

- Diagnosis Steps:

  • Checked pod resource requests and found they were too high for the available nodes.

  • Observed that scaling metrics showed no immediate response, and pods remained in a Pending state.

Root Cause: Resource requests were misconfigured, leading to a mismatch between node capacity and pod requirements.

- Fix/Workaround:

  • Reduced resource requests to better align with the available cluster resources.

  • Set resource limits more carefully based on load testing.

- Lessons Learned:

  Ensure that resource requests are configured properly to match the actual load requirements.

- How to Avoid:

  • Perform resource profiling and benchmarking before setting resource requests and limits.

  • Use metrics-based scaling strategies to adjust resources dynamically.

Scenario #442: Unexpected Pod Termination Due to Scaling Policy

- Category: Scaling & Load

- Environment: Kubernetes v1.23, Google Cloud

- Summary: Pods were unexpectedly terminated during scale-down due to aggressive scaling policies.

- What Happened:Scaling policy was too aggressive, and pods were removed even though they were still handling active traffic.

- Diagnosis Steps:

 • Reviewed scaling policy logs and found that the scaleDown strategy was too aggressive.

 • Metrics indicated that pods were removed before traffic spikes subsided.

 Root Cause: Aggressive scale-down policies without sufficient cool-down periods.

- Fix/Workaround:

 • Adjusted the scaleDown stabilization window and added buffer periods before termination.

 • Revisited scaling policy settings to ensure more balanced scaling.

- Lessons Learned:

 Scaling down should be done with more careful consideration, allowing for cool-down periods.

- How to Avoid:

 • Implement soft termination strategies to avoid premature pod removal.

 • Adjust the cool-down period in scale-down policies.

Scenario #443: Unstable Load Balancing During Scaling Events

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Azure AKS

- Summary: Load balancing issues surfaced during scaling, leading to uneven distribution of traffic.

- What Happened:As new pods were scaled up, traffic was not distributed evenly across them, causing some pods to be overwhelmed while others were underutilized.

- Diagnosis Steps:

• Investigated the load balancing configuration and found that the load balancer didn't adapt quickly to scaling changes.

• Found that new pods were added to the backend pool but not evenly distributed.

Root Cause: Load balancer misconfiguration, leading to uneven traffic distribution during scale-up events.

- Fix/Workaround:

• Reconfigured the load balancer to rebalance traffic more efficiently after scaling events.

• Adjusted readiness and liveness probes to allow new pods to join the pool smoothly.

- Lessons Learned:

Load balancers must be configured to dynamically adjust during scaling events.

- How to Avoid:

• Test and optimize load balancing settings in relation to pod scaling.

• Use health checks to ensure new pods are properly integrated into the load balancing pool.

Scenario #444: Autoscaling Ignored Due to Resource Quotas

- Category: Scaling & Load

- Environment: Kubernetes v1.26, IBM Cloud

- Summary: Resource quotas prevented autoscaling from triggering despite high load.

- What Happened:Although resource usage was high, autoscaling did not trigger because the namespace resource quota was already close to being exceeded.

- Diagnosis Steps:

• Reviewed quota settings and found that they limited pod creation in the namespace.

• Verified that resource usage exceeded limits, blocking new pod scaling.

Root Cause: Resource quotas in place blocked the creation of new pods, preventing autoscaling from responding.

- Fix/Workaround:

• Adjusted resource quotas to allow more flexible scaling.

• Implemented dynamic resource quota adjustments based on actual usage.

- Lessons Learned:

Resource quotas must be considered when designing autoscaling policies.

- How to Avoid:

• Regularly review and adjust resource quotas to allow for scaling flexibility.

• Monitor resource usage to ensure that quotas are not limiting necessary scaling.

Scenario #445: Delayed Scaling Response to Traffic Spike

- Category: Scaling & Load

- Environment: Kubernetes v1.24, GCP

- Summary: Scaling took too long to respond during a traffic spike, leading to degraded service.

- What Happened:Traffic surged unexpectedly, but the Horizontal Pod Autoscaler (HPA) was slow to scale up, leading to service delays.

- Diagnosis Steps:

• Reviewed HPA logs and found that the scaling threshold was too high for the initial traffic spike.

• Found that scaling policies were tuned for slower load increases, not sudden spikes.

Root Cause: Autoscaling thresholds were not tuned for quick response during traffic bursts.

- Fix/Workaround:

  • Lowered scaling thresholds to trigger scaling faster.
  • Used burst metrics for quicker scaling decisions.

- Lessons Learned:
  Autoscaling policies should be tuned for fast responses to sudden traffic spikes.

- How to Avoid:

  • Implement adaptive scaling thresholds based on traffic patterns.
  • Use real-time metrics to respond to sudden traffic bursts.

Scenario #446: CPU Utilization-Based Scaling Did Not Trigger for High Memory Usage

- Category: Scaling & Load

- Environment: Kubernetes v1.22, Azure AKS

- Summary: Scaling based on CPU utilization did not trigger when the issue was related to high memory usage.

- What Happened:Despite high memory usage, CPU-based scaling did not trigger any scaling events, causing performance degradation.

- Diagnosis Steps:

• Analyzed pod metrics and found that memory was saturated while CPU utilization was low.

• Checked HPA configuration, which was set to trigger based on CPU metrics, not memory.

Root Cause: Autoscaling was configured to use CPU utilization, not accounting for memory usage.

- Fix/Workaround:

• Configured HPA to also consider memory usage as a scaling metric.

• Adjusted scaling policies to scale pods based on both CPU and memory utilization.

- Lessons Learned:

Autoscaling should consider multiple resource metrics based on application needs.

- How to Avoid:

• Regularly assess the right metrics to base autoscaling decisions on.

• Tune autoscaling policies for the resource most affected during high load.

Scenario #447: Inefficient Horizontal Scaling of StatefulSets

- Category: Scaling & Load

- Environment: Kubernetes v1.25, GKE

- Summary: Horizontal scaling of StatefulSets was inefficient due to StatefulSet's inherent limitations.

- What Happened:Scaling horizontally caused issues with pod state and data integrity, as StatefulSet is not designed for horizontal scaling in certain scenarios.

- Diagnosis Steps:

  • Found that scaling horizontally caused pods to be spread across multiple nodes, breaking data consistency.

  • StatefulSet's lack of support for horizontal scaling led to instability.

  Root Cause: Misuse of StatefulSet for workloads that required horizontal scaling.

- Fix/Workaround:

  • Switched to a Deployment with persistent volumes, which better supported horizontal scaling for the workload.

  • Used StatefulSets only for workloads that require persistent state and stable network identities.

- Lessons Learned:

  StatefulSets are not suitable for all workloads, particularly those needing efficient horizontal scaling.

- How to Avoid:

  • Use StatefulSets only when necessary for specific use cases.

  • Consider alternative Kubernetes resources for scalable, stateless workloads.

Scenario #448: Autoscaler Skipped Scaling Events Due to Flaky Metrics

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Autoscaler skipped scaling events due to unreliable metrics from external monitoring tools.

- What Happened:Metrics from external monitoring systems were inconsistent, causing scaling decisions to be missed.

- Diagnosis Steps:

  • Checked the external monitoring tool integration with Kubernetes metrics and found data inconsistencies.

  • Discovered missing or inaccurate metrics led to missed scaling events.

  Root Cause: Unreliable third-party monitoring tool integration with Kubernetes.

- Fix/Workaround:

  • Switched to using native Kubernetes metrics for autoscaling decisions.

  • Ensured that metrics from third-party tools were properly validated before being used in autoscaling.

- Lessons Learned:

  Use native Kubernetes metrics where possible for more reliable autoscaling.

- How to Avoid:

  • Use built-in Kubernetes metrics server and Prometheus for reliable monitoring.

  • Validate third-party monitoring integrations to ensure accurate data.

Scenario #449: Delayed Pod Creation Due to Node Affinity Misconfigurations

- Category: Scaling & Load

- Environment: Kubernetes v1.24, Google Cloud

- Summary: Pods were delayed in being created due to misconfigured node affinity rules during scaling events.

- What Happened:Node affinity rules were too strict, leading to delays in pod scheduling when scaling up.

- Diagnosis Steps:

  • Reviewed node affinity rules and found they were unnecessarily restricting pod scheduling.

  • Observed that pods were stuck in the Pending state.

  Root Cause: Overly restrictive node affinity rules caused delays in pod scheduling.

- Fix/Workaround:

  • Loosened node affinity rules to allow more flexible scheduling.

• Used affinity rules more suited for scaling scenarios.

- Lessons Learned:

Node affinity must be carefully designed to allow for scaling flexibility.

- How to Avoid:

• Test affinity rules in scaling scenarios to ensure they don't block pod scheduling.

• Ensure that affinity rules are aligned with scaling requirements.

Scenario #450: Excessive Scaling During Short-Term Traffic Spikes

- Category: Scaling & Load

- Environment: Kubernetes v1.25, AWS EKS

- Summary: Autoscaling triggered excessive scaling during short-term traffic spikes, leading to unnecessary resource usage.

- What Happened:Autoscaler responded too aggressively to short bursts of traffic, over-provisioning resources.

- Diagnosis Steps:

• Analyzed autoscaler logs and found it responded to brief traffic spikes with unnecessary scaling.

• Metrics confirmed that scaling decisions were based on short-lived traffic spikes.

Root Cause: Autoscaler was too sensitive to short-term traffic fluctuations.

- Fix/Workaround:

  • Adjusted scaling policies to better handle short-term traffic spikes.

  • Implemented rate-limiting for scaling events.

- Lessons Learned:

  Autoscaling should account for long-term trends and ignore brief, short-lived spikes.

- How to Avoid:

  • Use cooldown periods or smoothing algorithms to prevent scaling from reacting to short-lived fluctuations.

  • Tune autoscaling policies based on long-term traffic patterns.

Scenario #451: Inconsistent Scaling Due to Misconfigured Horizontal Pod Autoscaler

- Category: Scaling & Load

- Environment: Kubernetes v1.26, Azure AKS

- Summary: Horizontal Pod Autoscaler (HPA) inconsistently scaled pods based on incorrect metric definitions.

- What Happened:HPA failed to scale up correctly because it was configured to trigger based on custom metrics, but the metric source was unreliable.

- Diagnosis Steps:

• Reviewed HPA configuration and identified incorrect metric configuration.

• Logs showed HPA was relying on a custom metric, which sometimes reported outdated or missing data.

Root Cause: Misconfigured custom metrics in the HPA setup, leading to inconsistent scaling decisions.

- Fix/Workaround:

• Switched to using Kubernetes-native CPU and memory metrics for autoscaling.

• Improved the reliability of the custom metrics system by implementing fallback mechanisms.

- Lessons Learned:

Custom metrics should be tested for reliability before being used in autoscaling decisions.

- How to Avoid:

• Regularly monitor and validate the health of custom metrics.

• Use native Kubernetes metrics for critical scaling decisions when possible.

Scenario #452: Load Balancer Overload After Quick Pod Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: Load balancer failed to distribute traffic effectively after a large pod scaling event, leading to overloaded pods.

- What Happened:Pods were scaled up quickly, but the load balancer did not reassign traffic in a timely manner, causing some pods to receive too much traffic while others were underutilized.

- Diagnosis Steps:

• Investigated the load balancer configuration and found that traffic routing did not adjust immediately after the scaling event.

• Noticed uneven distribution of traffic in the load balancer dashboard.

Root Cause: Load balancer was not properly configured to dynamically rebalance traffic after pod scaling.

- Fix/Workaround:

• Reconfigured the load balancer to automatically adjust traffic distribution after pod scaling events.

• Implemented health checks to ensure that only fully initialized pods received traffic.

- Lessons Learned:

Load balancers must be able to react quickly to changes in the backend pool after scaling.

- How to Avoid:

• Use auto-scaling triggers that also adjust load balancer settings dynamically.

• Implement smarter traffic management for faster pod scale-up transitions.

Scenario #453: Autoscaling Failed During Peak Traffic Periods

- Category: Scaling & Load

- Environment: Kubernetes v1.24, AWS EKS

- Summary: Autoscaling was ineffective during peak traffic periods, leading to degraded performance.

- What Happened:Although traffic spikes were detected, the Horizontal Pod Autoscaler (HPA) failed to scale up the required number of pods in time.

- Diagnosis Steps:

• Analyzed HPA metrics and scaling logs, which revealed that the scaling trigger was set with a high threshold.

• Traffic metrics indicated that the spike was gradual but persistent, triggering a delayed scaling response.

Root Cause: Autoscaling thresholds were not sensitive enough to handle gradual, persistent traffic spikes.

- Fix/Workaround:

• Lowered the scaling thresholds to respond more quickly to persistent traffic increases.

• Implemented more granular scaling rules based on time-based patterns.

- Lessons Learned:

Autoscaling policies need to be tuned to handle gradual traffic increases, not just sudden bursts.

- How to Avoid:

• Implement time-based or persistent traffic-based autoscaling rules.

• Regularly monitor and adjust scaling thresholds based on actual traffic patterns.

Scenario #454: Insufficient Node Resources During Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.23, IBM Cloud

- Summary: Node resources were insufficient during scaling, leading to pod scheduling failures.

- What Happened:Pods failed to scale up because there were not enough resources on existing nodes to accommodate them.

- Diagnosis Steps:

• Checked node resource availability and found that there were insufficient CPU or memory resources for the new pods.

• Horizontal scaling was triggered, but node resource limitations prevented pod scheduling.

Root Cause: Node resources were exhausted, causing pod placement to fail during scaling.

- Fix/Workaround:

• Increased the resource limits on existing nodes.

• Implemented Cluster Autoscaler to add more nodes when resources are insufficient.

- Lessons Learned:

Ensure that the cluster has sufficient resources or can scale horizontally when pod demands increase.

- How to Avoid:

• Use Cluster Autoscaler or manage node pool resources dynamically based on scaling needs.

• Regularly monitor resource utilization to avoid saturation during scaling events.

Scenario #455: Unpredictable Pod Scaling During Cluster Autoscaler Event

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: Pod scaling was unpredictable during a Cluster Autoscaler event due to a sudden increase in node availability.

- What Happened:When Cluster Autoscaler added new nodes to the cluster, the autoscaling process became erratic as new pods were scheduled in unpredictable order.

- Diagnosis Steps:

  • Analyzed scaling logs and found that new nodes were provisioned, but pod scheduling was not coordinated well with available node resources.

  • Observed that new pods were not placed efficiently on the newly provisioned nodes.

  Root Cause: Cluster Autoscaler was adding new nodes too quickly without proper scheduling coordination.

- Fix/Workaround:

  • Adjusted Cluster Autoscaler settings to delay node addition during scaling events.

  • Tweaked pod scheduling policies to ensure new pods were placed on the most appropriate nodes.

- Lessons Learned:

  Cluster Autoscaler should work more harmoniously with pod scheduling to ensure efficient scaling.

- How to Avoid:

  • Fine-tune Cluster Autoscaler settings to prevent over-rapid node provisioning.

  • Use more advanced scheduling policies to manage pod placement efficiently.

Scenario #456: CPU Resource Over-Commitment During Scale-Up

- Category: Scaling & Load

- Environment: Kubernetes v1.23, Azure AKS

- Summary: During a scale-up event, CPU resources were over-committed, causing pod performance degradation.

- What Happened:When scaling up, CPU resources were over-allocated to new pods, leading to performance degradation as existing pods had to share CPU cores.

- Diagnosis Steps:

  • Checked CPU resource allocation and found that the new pods had been allocated higher CPU shares than the existing pods, causing resource contention.

  • Observed significant latency and degraded performance in the cluster.

  Root Cause: Resource allocation was not adjusted for existing pods, causing CPU contention during scale-up.

- Fix/Workaround:

  • Adjusted the CPU resource limits and requests for new pods to avoid over-commitment.

  • Implemented resource isolation policies to prevent CPU contention.

- Lessons Learned:

Proper resource allocation strategies are essential during scale-up to avoid resource contention.

- How to Avoid:

  • Use CPU and memory limits to avoid resource over-commitment.

  • Implement resource isolation techniques like CPU pinning or dedicated nodes for specific workloads.

Scenario #457: Failure to Scale Due to Horizontal Pod Autoscaler Anomaly

- Category: Scaling & Load

- Environment: Kubernetes v1.22, AWS EKS

- Summary: Horizontal Pod Autoscaler (HPA) failed to scale up due to a temporary anomaly in the resource metrics.

- What Happened:HPA failed to trigger a scale-up action during a high traffic period because resource metrics were temporarily inaccurate.

- Diagnosis Steps:

  • Checked metrics server logs and found that there was a temporary issue with the metric collection process.

  • Metrics were not properly reflecting the true resource usage due to a short-lived anomaly.

Root Cause: Temporary anomaly in the metric collection system led to inaccurate scaling decisions.

- Fix/Workaround:

  • Implemented a fallback mechanism to trigger scaling based on last known good metrics.

  • Used a more robust monitoring system to track resource usage in real time.

- Lessons Learned:

  Autoscalers should have fallback mechanisms for temporary metric anomalies.

- How to Avoid:

  • Set up fallback mechanisms and monitoring alerts to handle metric inconsistencies.

  • Regularly test autoscaling responses to ensure reliability.

Scenario #458: Memory Pressure Causing Slow Pod Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, IBM Cloud

- Summary: Pod scaling was delayed due to memory pressure in the cluster, causing performance bottlenecks.

- What Happened:Pods scaled slowly during high memory usage periods because of memory pressure on existing nodes.

- Diagnosis Steps:

  • Checked node metrics and found that there was significant memory pressure on the nodes, delaying pod scheduling.

  • Memory was allocated too heavily to existing pods, leading to delays in new pod scheduling.

  Root Cause: High memory pressure on nodes, causing delays in pod scaling.

- Fix/Workaround:

  • Increased the memory available on nodes to alleviate pressure.

  • Used resource requests and limits more conservatively to ensure proper memory allocation.

- Lessons Learned:

  Node memory usage must be managed carefully during scaling events to avoid delays.

- How to Avoid:

  • Monitor node memory usage and avoid over-allocation of resources.

  • Use memory-based autoscaling to ensure adequate resources are available during traffic spikes.

Scenario #459: Node Over-Provisioning During Cluster Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: Nodes were over-provisioned, leading to unnecessary resource wastage during scaling.

- What Happened:Cluster Autoscaler added more nodes than necessary during scaling events, leading to resource wastage.

- Diagnosis Steps:

  • Reviewed the scaling logic and determined that the Autoscaler was provisioning more nodes than required to handle the traffic load.

  • Node usage data indicated that several nodes remained underutilized.

  Root Cause: Over-provisioning by the Cluster Autoscaler due to overly conservative scaling settings.

- Fix/Workaround:

  • Fine-tuned Cluster Autoscaler settings to scale nodes more precisely based on actual usage.

  • Implemented tighter limits on node scaling thresholds.

- Lessons Learned:

  Autoscaler settings must be precise to avoid over-provisioning and resource wastage.

- How to Avoid:

• Regularly monitor node usage and adjust scaling thresholds.

• Implement smarter autoscaling strategies that consider the actual resource demand.

Scenario #460: Autoscaler Fails to Handle Node Termination Events Properly

- Category: Scaling & Load

- Environment: Kubernetes v1.26, Azure AKS

- Summary: Autoscaler did not handle node termination events properly, leading to pod disruptions.

- What Happened:When nodes were terminated due to failure or maintenance, the autoscaler failed to replace them quickly enough, leading to pod disruption.

- Diagnosis Steps:

• Checked autoscaler logs and found that termination events were not triggering prompt scaling actions.

• Node failure events showed that the cluster was slow to react to node loss.

Root Cause: Autoscaler was not tuned to respond quickly enough to node terminations.

- Fix/Workaround:

• Configured the autoscaler to prioritize the immediate replacement of terminated nodes.

• Enhanced the health checks to better detect node failures.

- Lessons Learned:

Autoscalers must be configured to respond quickly to node failure and termination events.

- How to Avoid:

• Implement tighter integration between node health checks and autoscaling triggers.

• Ensure autoscaling settings prioritize quick recovery from node failures.

Scenario #461: Node Failure During Pod Scaling Up

- Category: Scaling & Load

- Environment: Kubernetes v1.25, AWS EKS

- Summary: Scaling up pods failed when a node was unexpectedly terminated, preventing proper pod scheduling.

- What Happened:During an autoscaling event, a node was unexpectedly terminated due to cloud infrastructure issues. This caused new pods to fail scheduling as no available node had sufficient resources.

- Diagnosis Steps:

• Checked the node status and found that the node had been terminated by AWS.

• Observed that there were no available nodes with the required resources for new pods.

Root Cause: Unexpected node failure during the scaling process.

- Fix/Workaround:

• Configured the Cluster Autoscaler to provision more nodes and preemptively account for potential node failures.

• Ensured the cloud provider's infrastructure health was regularly monitored.

- Lessons Learned:

Autoscaling should anticipate infrastructure issues such as node failure to avoid disruptions.

- How to Avoid:

• Set up proactive monitoring for cloud infrastructure and integrate with Kubernetes scaling mechanisms.

• Ensure Cluster Autoscaler is tuned to handle unexpected node failures quickly.

Scenario #462: Unstable Scaling During Traffic Spikes

- Category: Scaling & Load

- Environment: Kubernetes v1.26, Azure AKS

- Summary: Pod scaling became unstable during traffic spikes due to delayed scaling responses.

- What Happened:During high-traffic periods, HPA (Horizontal Pod Autoscaler) did not scale pods fast enough, leading to slow response times.

- Diagnosis Steps:

  • Reviewed HPA logs and metrics and discovered scaling triggers were based on 5-minute intervals, which caused delayed reactions to rapid traffic increases.

  • Observed increased latency and 504 Gateway Timeout errors.

  Root Cause: Autoscaler was not responsive enough to quickly scale up based on rapidly changing traffic.

- Fix/Workaround:

  • Adjusted the scaling policy to use smaller time intervals for triggering scaling.

  • Introduced custom metrics to scale pods based on response times and traffic patterns.

- Lessons Learned:

  Autoscaling should be sensitive to real-time traffic patterns and latency.

- How to Avoid:

  • Tune HPA to scale more aggressively during traffic spikes.

  • Use more advanced metrics like response time, rather than just CPU and memory, for autoscaling decisions.

Scenario #463: Insufficient Node Pools During Sudden Pod Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, Google Cloud

- Summary: Insufficient node pool capacity caused pod scheduling failures during sudden scaling events.

- What Happened:During a sudden traffic surge, the Horizontal Pod Autoscaler (HPA) scaled the pods, but there weren't enough nodes available to schedule the new pods.

- Diagnosis Steps:

  • Checked the available resources on the nodes and found that node pools were insufficient to accommodate the newly scaled pods.

  • Cluster logs revealed the autoscaler did not add more nodes promptly.

  Root Cause: Node pool capacity was insufficient, and the autoscaler did not scale the cluster quickly enough.

- Fix/Workaround:

  • Expanded node pool size to accommodate more pods.

  • Adjusted autoscaling policies to trigger faster node provisioning during scaling events.

- Lessons Learned:

Autoscaling node pools must be able to respond quickly during sudden traffic surges.

- How to Avoid:

• Pre-configure node pools to handle expected traffic growth, and ensure autoscalers are tuned to scale quickly.

Scenario #464: Latency Spikes During Horizontal Pod Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.25, IBM Cloud

- Summary: Latency spikes occurred during horizontal pod scaling due to inefficient pod distribution.

- What Happened:Horizontal pod scaling caused latency spikes as the traffic was unevenly distributed between pods, some of which were underutilized while others were overloaded.

- Diagnosis Steps:

• Reviewed traffic distribution and pod scheduling, which revealed that the load balancer did not immediately update routing configurations.

• Found that newly scaled pods were not receiving traffic promptly.

Root Cause: Delayed update in load balancer routing configuration after scaling.

- Fix/Workaround:

• Configured load balancer to refresh routing rules as soon as new pods were scaled up.

• Implemented readiness probes to ensure that only fully initialized pods were exposed to traffic.

- Lessons Learned:

Load balancer reconfiguration must be synchronized with pod scaling events.

- How to Avoid:

• Use automatic load balancer updates during scaling events.

• Configure readiness probes to ensure proper pod initialization before they handle traffic.

Scenario #465: Resource Starvation During Infrequent Scaling Events

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: During infrequent scaling events, resource starvation occurred due to improper resource allocation.

- What Happened:Infrequent scaling triggered by traffic bursts led to resource starvation on nodes, preventing pod scheduling.

- Diagnosis Steps:

  • Analyzed the scaling logs and found that resource allocation during scaling events was inadequate to meet the traffic demands.

  • Observed that resource starvation was particularly high for CPU and memory during scaling.

  Root Cause: Improper resource allocation strategy during pod scaling events.

- Fix/Workaround:

  • Adjusted resource requests and limits to better reflect the actual usage during scaling events.

  • Increased node pool size to provide more headroom during burst scaling.

- Lessons Learned:

  Resource requests must align with actual usage during scaling events to prevent starvation.

- How to Avoid:

  • Implement more accurate resource monitoring and adjust scaling policies based on real traffic usage patterns.

Scenario #466: Autoscaler Delayed Reaction to Load Decrease

- Category: Scaling & Load

- Environment: Kubernetes v1.22, Google Cloud

- Summary: The autoscaler was slow to scale down after a drop in traffic, causing resource wastage.

- What Happened:After a traffic drop, the Horizontal Pod Autoscaler (HPA) did not scale down quickly enough, leading to resource wastage.

- Diagnosis Steps:

  • Checked autoscaler logs and observed that it was still running extra pods even after traffic had reduced significantly.

  • Resource metrics indicated that there were idle pods consuming CPU and memory unnecessarily.

  Root Cause: HPA configuration was not tuned to respond quickly enough to a traffic decrease.

- Fix/Workaround:

  • Reduced the cooldown period in the HPA configuration to make it more responsive to traffic decreases.

  • Set resource limits to better reflect current traffic levels.

- Lessons Learned:

  Autoscalers should be configured with sensitivity to both traffic increases and decreases.

- How to Avoid:

• Tune HPA with shorter cooldown periods for faster scaling adjustments during both traffic surges and drops.

• Monitor traffic trends and adjust scaling policies accordingly.

Scenario #467: Node Resource Exhaustion Due to High Pod Density

- Category: Scaling & Load

- Environment: Kubernetes v1.24, Azure AKS

- Summary: Node resource exhaustion occurred when too many pods were scheduled on a single node, leading to instability.

- What Happened:During scaling events, pods were scheduled too densely on a single node, causing resource exhaustion and instability.

- Diagnosis Steps:

• Reviewed node resource utilization, which showed that the CPU and memory were maxed out on the affected nodes.

• Pods were not distributed evenly across the cluster.

Root Cause: Over-scheduling pods on a single node during scaling events caused resource exhaustion.

- Fix/Workaround:

• Adjusted pod affinity rules to distribute pods more evenly across the cluster.

• Increased the number of nodes available to handle the pod load more effectively.

- Lessons Learned:

Resource exhaustion can occur if pod density is not properly managed across nodes.

- How to Avoid:

• Use pod affinity and anti-affinity rules to control pod placement during scaling events.

• Ensure that the cluster has enough nodes to handle the pod density.

Scenario #468: Scaling Failure Due to Node Memory Pressure

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: Pod scaling failed due to memory pressure on nodes, preventing new pods from being scheduled.

- What Happened:Memory pressure on nodes prevented new pods from being scheduled, even though scaling events were triggered.

- Diagnosis Steps:

• Checked memory utilization and found that nodes were operating under high memory pressure, causing scheduling failures.

• Noticed that pod resource requests were too high for the available memory.

Root Cause: Insufficient memory resources on nodes to accommodate the newly scaled pods.

- Fix/Workaround:

• Increased memory resources on nodes and adjusted pod resource requests to better match available resources.

• Implemented memory-based autoscaling to handle memory pressure better during scaling events.

- Lessons Learned:

Memory pressure must be monitored and managed effectively during scaling events to avoid pod scheduling failures.

- How to Avoid:

• Ensure nodes have sufficient memory available, and use memory-based autoscaling.

• Implement tighter control over pod resource requests and limits.

Scenario #469: Scaling Latency Due to Slow Node Provisioning

- Category: Scaling & Load

- Environment: Kubernetes v1.26, IBM Cloud

- Summary: Pod scaling was delayed due to slow node provisioning during cluster scaling events.

- What Happened:When the cluster scaled up, node provisioning was slow, causing delays in pod scheduling and a degraded user experience.

- Diagnosis Steps:

• Reviewed cluster scaling logs and found that the time taken for new nodes to become available was too long.

• Latency metrics showed that the pods were not ready to handle traffic in time.

Root Cause: Slow node provisioning due to cloud infrastructure limitations.

- Fix/Workaround:

• Worked with the cloud provider to speed up node provisioning times.

• Used preemptible nodes to quickly handle scaling demands during traffic spikes.

- Lessons Learned:

Node provisioning speed can have a significant impact on scaling performance.

- How to Avoid:

• Work closely with the cloud provider to optimize node provisioning speed.

• Use faster provisioning options like preemptible nodes for scaling events.

Scenario #470: Slow Scaling Response Due to Insufficient Metrics Collection

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: The autoscaling mechanism responded slowly to traffic changes because of insufficient metrics collection.

- What Happened:The Horizontal Pod Autoscaler (HPA) failed to trigger scaling events quickly enough due to missing or outdated metrics, resulting in delayed scaling during traffic spikes.

- Diagnosis Steps:

  • Checked HPA logs and observed that the scaling behavior was delayed, even though CPU and memory usage had surged.

  • Discovered that custom metrics used by HPA were not being collected in real-time.

  Root Cause: Missing or outdated custom metrics, which slowed down autoscaling.

- Fix/Workaround:

  • Updated the metric collection to use real-time data, reducing the delay in scaling actions.

  • Implemented a more frequent metric scraping interval to improve responsiveness.

- Lessons Learned:

  Autoscaling depends heavily on accurate and up-to-date metrics.

- How to Avoid:

• Ensure that all required metrics are collected in real-time for responsive scaling.

• Set up alerting for missing or outdated metrics.

Scenario #471: Node Scaling Delayed Due to Cloud Provider API Limits

- Category: Scaling & Load

- Environment: Kubernetes v1.24, Google Cloud

- Summary: Node scaling was delayed because the cloud provider's API rate limits were exceeded, preventing automatic node provisioning.

- What Happened:During a scaling event, the Cloud Provider API rate limits were exceeded, and the Kubernetes Cluster Autoscaler failed to provision new nodes, causing pod scheduling delays.

- Diagnosis Steps:

• Checked the autoscaler logs and found that the scaling action was queued due to API rate limit restrictions.

• Observed that new nodes were not added promptly, leading to pod scheduling failures.

Root Cause: Exceeded API rate limits for cloud infrastructure.

- Fix/Workaround:

• Worked with the cloud provider to increase API rate limits.

• Configured autoscaling to use multiple API keys to distribute the API requests and avoid hitting rate limits.

- Lessons Learned:

Cloud infrastructure APIs can have rate limits that may affect scaling.

- How to Avoid:

• Monitor cloud API rate limits and set up alerting for approaching thresholds.

• Use multiple API keys for autoscaling operations to avoid hitting rate limits.

Scenario #472: Scaling Overload Due to High Replica Count

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Azure AKS

- Summary: Pod scaling led to resource overload on nodes due to an excessively high replica count.

- What Happened:A configuration error caused the Horizontal Pod Autoscaler (HPA) to scale up to an unusually high replica count, leading to CPU and memory overload on the nodes.

- Diagnosis Steps:

• Checked HPA configuration and found that the scaling target was incorrectly set to a high replica count.

• Monitored node resources, which were exhausted due to the large number of pods.

Root Cause: Misconfigured replica count in the autoscaler configuration.

- Fix/Workaround:

• Adjusted the replica scaling thresholds in the HPA configuration.

• Limited the maximum replica count to avoid overload.

- Lessons Learned:

Scaling should always have upper limits to prevent resource exhaustion.

- How to Avoid:

• Set upper limits for pod replicas and ensure that scaling policies are appropriate for the available resources.

Scenario #473: Failure to Scale Down Due to Persistent Idle Pods

- Category: Scaling & Load

- Environment: Kubernetes v1.24, IBM Cloud

- Summary: Pods failed to scale down during low traffic periods, leading to idle resources consuming cluster capacity.

- What Happened:During low traffic periods, the Horizontal Pod Autoscaler (HPA) failed to scale down pods because some pods were marked as "not ready" but still consuming resources.

- Diagnosis Steps:

• Checked HPA configuration and found that some pods were stuck in a “not ready” state.

• Identified that these pods were preventing the autoscaler from scaling down.

Root Cause: Pods marked as “not ready” were still consuming resources, preventing autoscaling.

- Fix/Workaround:

• Updated the readiness probe configuration to ensure pods were correctly marked as ready or not based on their actual state.

• Configured the HPA to scale down based on actual pod readiness.

- Lessons Learned:

Autoscaling can be disrupted by incorrectly configured readiness probes or failing pods.

- How to Avoid:

• Regularly review and adjust readiness probes to ensure they reflect the actual health of pods.

• Set up alerts for unresponsive pods that could block scaling.

Scenario #474: Load Balancer Misrouting After Pod Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.26, AWS EKS

- Summary: The load balancer routed traffic unevenly after scaling up, causing some pods to become overloaded.

- What Happened:After pod scaling, the load balancer did not immediately update routing rules, leading to uneven traffic distribution. Some pods became overloaded, while others were underutilized.

- Diagnosis Steps:

  • Checked load balancer configuration and found that it had not updated its routing rules after pod scaling.

  • Observed uneven traffic distribution on the affected pods.

  Root Cause: Delayed load balancer reconfiguration after scaling events.

- Fix/Workaround:

  • Configured the load balancer to refresh routing rules dynamically during pod scaling events.

  • Ensured that only ready and healthy pods were included in the load balancer's routing pool.

- Lessons Learned:

  Load balancers must be synchronized with pod scaling events to ensure even traffic distribution.

- How to Avoid:

• Automate load balancer rule updates during scaling events.

• Integrate health checks and readiness probes to ensure only available pods handle traffic.

Scenario #475: Cluster Autoscaler Not Triggering Under High Load

- Category: Scaling & Load

- Environment: Kubernetes v1.22, Google Cloud

- Summary: The Cluster Autoscaler failed to trigger under high load due to misconfiguration in resource requests.

- What Happened:Despite a high load on the cluster, the Cluster Autoscaler did not trigger additional nodes due to misconfigured resource requests for pods.

- Diagnosis Steps:

• Reviewed autoscaler logs and resource requests, and discovered that pods were requesting more resources than available on the nodes.

• Resource requests exceeded available node capacity, but the autoscaler did not respond appropriately.

Root Cause: Misconfigured resource requests for pods, leading to poor autoscaler behavior.

- Fix/Workaround:

  • Adjusted resource requests and limits to match node capacity.

  • Tuned the Cluster Autoscaler to scale more aggressively during high load situations.

- Lessons Learned:

  Proper resource requests are critical for effective autoscaling.

- How to Avoid:

  • Continuously monitor and adjust resource requests based on actual usage patterns.

  • Use autoscaling metrics that consider both resource usage and load.

Scenario #476: Autoscaling Slow Due to Cloud Provider API Delay

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Azure AKS

- Summary: Pod scaling was delayed due to cloud provider API delays during scaling events.

- What Happened:Scaling actions were delayed because the cloud provider API took longer than expected to provision new resources, affecting pod scheduling.

- Diagnosis Steps:

• Checked the scaling event logs and found that new nodes were being provisioned slowly due to API rate limiting.

• Observed delayed pod scheduling as a result of slow node availability.

Root Cause: Slow cloud provider API response times and rate limiting.

- Fix/Workaround:

• Worked with the cloud provider to optimize node provisioning time.

• Increased API limits to accommodate the scaling operations.

- Lessons Learned:

Cloud infrastructure API response time can impact scaling performance.

- How to Avoid:

• Ensure that the cloud provider API is optimized and scalable.

• Work with the provider to avoid rate limits during scaling events.

Scenario #477: Over-provisioning Resources During Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, IBM Cloud

- Summary: During a scaling event, resources were over-provisioned, causing unnecessary resource consumption and cost.

- What Happened:During scaling, the resources requested by pods were higher than needed, leading to over-provisioning and unnecessary resource consumption.

- Diagnosis Steps:

  • Reviewed pod resource requests and limits, finding that they were set higher than the actual usage.

  • Observed higher-than-expected costs due to over-provisioning.

  Root Cause: Misconfigured pod resource requests and limits during scaling.

- Fix/Workaround:

  • Reduced resource requests and limits to more closely match actual usage patterns.

  • Enabled auto-scaling of resource limits based on traffic patterns.

- Lessons Learned:

  Over-provisioning can lead to resource wastage and increased costs.

- How to Avoid:

  • Fine-tune resource requests and limits based on historical usage and traffic patterns.

  • Use monitoring tools to track resource usage and adjust requests accordingly.

Scenario #478: Incorrect Load Balancer Configuration After Node Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: After node scaling, the load balancer failed to distribute traffic correctly due to misconfigured settings.

- What Happened:Scaling added new nodes, but the load balancer configuration was not updated correctly, leading to traffic being routed to the wrong nodes.

- Diagnosis Steps:

  • Checked the load balancer configuration and found that it was not dynamically updated after node scaling.

  • Traffic logs showed that certain nodes were not receiving traffic despite having available resources.

  Root Cause: Misconfigured load balancer settings after scaling.

- Fix/Workaround:

  • Updated load balancer settings to ensure they dynamically adjust based on node changes.

  • Implemented a health check system for nodes before routing traffic.

- Lessons Learned:

  Load balancers must adapt dynamically to node scaling events.

- How to Avoid:

  • Set up automation to update load balancer configurations during scaling events.

  • Regularly test load balancer reconfigurations.

Scenario #478: Incorrect Load Balancer Configuration After Node Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: After node scaling, the load balancer failed to distribute traffic correctly due to misconfigured settings.

- What Happened:Scaling added new nodes, but the load balancer configuration was not updated correctly, leading to traffic being routed to the wrong nodes.

- Diagnosis Steps:

  • Checked the load balancer configuration and found that it was not dynamically updated after node scaling.

  • Traffic logs showed that certain nodes were not receiving traffic despite having available resources.

  Root Cause: Misconfigured load balancer settings after scaling.

- Fix/Workaround:

  • Updated load balancer settings to ensure they dynamically adjust based on node changes.

  • Implemented a health check system for nodes before routing traffic.

- Lessons Learned:

Load balancers must adapt dynamically to node scaling events.

- How to Avoid:

  • Set up automation to update load balancer configurations during scaling events.
  • Regularly test load balancer reconfigurations.

Scenario #479: Autoscaling Disabled Due to Resource Constraints

- Category: Scaling & Load

- Environment: Kubernetes v1.22, AWS EKS

- Summary: Autoscaling was disabled due to resource constraints on the cluster.

- What Happened:During a traffic spike, autoscaling was unable to trigger because the cluster had insufficient resources to create new nodes.

- Diagnosis Steps:

  • Reviewed Cluster Autoscaler logs and found that the scaling attempt failed because there were not enough resources in the cloud to provision new nodes.
  • Observed that resource requests and limits on existing pods were high.

Root Cause: Cluster was running at full capacity, and the cloud provider could not provision additional resources.

- Fix/Workaround:

• Reduced resource requests and limits on existing pods.

• Requested additional capacity from the cloud provider to handle scaling operations.

- Lessons Learned:

Autoscaling is only effective if there are sufficient resources to provision new nodes.

- How to Avoid:

• Monitor available cluster resources and ensure that there is capacity for scaling events.

• Configure the Cluster Autoscaler to scale based on real-time resource availability.

Scenario #480: Resource Fragmentation Leading to Scaling Delays

- Category: Scaling & Load

- Environment: Kubernetes v1.24, Azure AKS

- Summary: Fragmentation of resources across nodes led to scaling delays as new pods could not be scheduled efficiently.

- What Happened:As the cluster scaled, resources were fragmented across nodes, and new pods couldn't be scheduled quickly due to uneven distribution of CPU and memory.

- Diagnosis Steps:

• Checked pod scheduling logs and found that new pods were not scheduled because of insufficient resources on existing nodes.

• Observed that resource fragmentation led to inefficient usage of available capacity.

Root Cause: Fragmented resources, where existing nodes had unused capacity but could not schedule new pods due to resource imbalances.

- Fix/Workaround:

• Enabled pod affinity and anti-affinity rules to ensure better distribution of pods across nodes.

• Reconfigured node selectors and affinity rules for optimal pod placement.

- Lessons Learned:

Resource fragmentation can slow down pod scheduling and delay scaling.

- How to Avoid:

• Implement better resource scheduling strategies using affinity and anti-affinity rules.

• Regularly monitor and rebalance resources across nodes to ensure efficient pod scheduling.

Scenario #481: Incorrect Scaling Triggers Due to Misconfigured Metrics Server

- Category: Scaling & Load

- Environment: Kubernetes v1.26, IBM Cloud

- Summary: The HPA scaled pods incorrectly because the metrics server was misconfigured, leading to wrong scaling triggers.

- What Happened:The Horizontal Pod Autoscaler (HPA) triggered scaling events based on inaccurate metrics from a misconfigured metrics server, causing pods to scale up and down erratically.

- Diagnosis Steps:

  • Reviewed HPA configuration and found that it was using incorrect metrics due to a misconfigured metrics server.

  • Observed fluctuations in pod replicas despite stable traffic and resource utilization.

  Root Cause: Misconfigured metrics server, providing inaccurate data for scaling.

- Fix/Workaround:

  • Corrected the metrics server configuration to ensure it provided accurate resource data.

  • Adjusted the scaling thresholds to be more aligned with actual traffic patterns.

- Lessons Learned:

  Accurate metrics are crucial for autoscaling to work effectively.

- How to Avoid:

  • Regularly audit metrics servers to ensure they are correctly collecting and reporting data.

  • Use redundancy in metrics collection to avoid single points of failure.

Scenario #482: Autoscaler Misconfigured with Cluster Network Constraints

- Category: Scaling & Load

- Environment: Kubernetes v1.25, Google Cloud

- Summary: The Cluster Autoscaler failed to scale due to network configuration constraints that prevented communication between nodes.

- What Happened:Cluster Autoscaler tried to add new nodes, but network constraints in the cluster configuration prevented nodes from communicating, causing scaling to fail.

- Diagnosis Steps:

  • Checked network logs and found that new nodes could not communicate with the existing cluster.

  • Found that the network policy or firewall rules were blocking traffic to new nodes.

  Root Cause: Misconfigured network policies or firewall rules preventing new nodes from joining the cluster.

- Fix/Workaround:

  • Adjusted network policies and firewall rules to allow communication between new and existing nodes.

  • Configured the autoscaler to take network constraints into account during scaling events.

- Lessons Learned:

Network constraints can block scaling operations, especially when adding new nodes.

- How to Avoid:

• Test and review network policies and firewall rules periodically to ensure new nodes can be integrated into the cluster.

• Ensure that scaling operations account for network constraints.

Scenario #483: Scaling Delays Due to Resource Quota Exhaustion

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Pod scaling was delayed due to exhausted resource quotas, preventing new pods from being scheduled.

- What Happened:When attempting to scale, the system could not schedule new pods because the resource quotas for the namespace were exhausted.

- Diagnosis Steps:

• Checked the resource quota settings for the namespace and confirmed that the available resource quota had been exceeded.

• Observed that scaling attempts were blocked as a result.

Root Cause: Resource quotas were not properly adjusted to accommodate dynamic scaling needs.

- Fix/Workaround:

  • Increased the resource quotas to allow for more pods and scaling capacity.
  • Reviewed and adjusted resource quotas to ensure they aligned with expected scaling behavior.

- Lessons Learned:
  Resource quotas must be dynamically adjusted to match scaling requirements.

- How to Avoid:

  • Monitor and adjust resource quotas regularly to accommodate scaling needs.
  • Set up alerting for approaching resource quota limits to avoid scaling issues.

Scenario #484: Memory Resource Overload During Scaling

- Category: Scaling & Load

- Environment: Kubernetes v1.24, Azure AKS

- Summary: Node memory resources were exhausted during a scaling event, causing pods to crash.

- What Happened:As the cluster scaled, nodes did not have enough memory resources to accommodate the new pods, causing the pods to crash and leading to high memory pressure.

- Diagnosis Steps:

• Checked pod resource usage and found that memory limits were exceeded, leading to eviction of pods.

• Observed that the scaling event did not consider memory usage in the node resource calculations.

Root Cause: Insufficient memory on nodes during scaling events, leading to pod crashes.

- Fix/Workaround:

• Adjusted pod memory requests and limits to avoid over-provisioning.

• Increased memory resources on the nodes to handle the scaled workload.

- Lessons Learned:

Memory pressure is a critical factor in scaling, and it should be carefully considered during node provisioning.

- How to Avoid:

• Monitor memory usage closely during scaling events.

• Ensure that scaling policies account for both CPU and memory resources.

Scenario #485: HPA Scaling Delays Due to Incorrect Metric Aggregation

- Category: Scaling & Load

- Environment: Kubernetes v1.26, Google Cloud

- Summary: HPA scaling was delayed due to incorrect aggregation of metrics, leading to slower response to traffic spikes.

- What Happened:The HPA scaled slowly because the metric server was aggregating metrics at an incorrect rate, delaying scaling actions.

- Diagnosis Steps:

  • Reviewed HPA and metrics server configuration, and found incorrect aggregation settings that slowed down metric reporting.

  • Observed that the scaling actions did not trigger as quickly as expected during traffic spikes.

  Root Cause: Incorrect metric aggregation settings in the metric server.

- Fix/Workaround:

  • Corrected the aggregation settings to ensure faster response times for scaling events.

  • Tuned the HPA configuration to react more quickly to traffic fluctuations.

- Lessons Learned:

  Accurate and timely metric aggregation is crucial for effective scaling.

- How to Avoid:

• Regularly review metric aggregation settings to ensure they support rapid scaling decisions.

• Set up alerting for scaling delays and metric anomalies.

Scenario #486: Scaling Causing Unbalanced Pods Across Availability Zones

- Category: Scaling & Load

- Environment: Kubernetes v1.25, AWS EKS

- Summary: Pods became unbalanced across availability zones during scaling, leading to higher latency for some traffic.

- What Happened:During scaling, the pod scheduler did not evenly distribute pods across availability zones, leading to pod concentration in one zone and increased latency in others.

- Diagnosis Steps:

• Reviewed pod placement logs and found that the scheduler was not balancing pods across zones as expected.

• Traffic logs showed increased latency in one of the availability zones.

Root Cause: Misconfigured affinity rules leading to unbalanced pod distribution.

- Fix/Workaround:

• Reconfigured pod affinity rules to ensure an even distribution across availability zones.

• Implemented anti-affinity rules to avoid overloading specific zones.

- Lessons Learned:

Proper pod placement is crucial for high availability and low latency.

- How to Avoid:

• Use affinity and anti-affinity rules to ensure even distribution across availability zones.

• Regularly monitor pod distribution and adjust scheduling policies as needed.

Scenario #487: Failed Scaling due to Insufficient Node Capacity for StatefulSets

- Category: Scaling & Load

- Environment: Kubernetes v1.23, AWS EKS

- Summary: Scaling failed because the node pool did not have sufficient capacity to accommodate new StatefulSets.

- What Happened:When trying to scale a StatefulSet, the system couldn't allocate enough resources on the available nodes, causing scaling to fail.

- Diagnosis Steps:

• Checked resource availability across nodes and found that there wasn’t enough storage or CPU capacity for StatefulSet pods.

• Observed that the cluster's persistent volume claims (PVCs) were causing resource constraints.

Root Cause: Inadequate resource allocation, particularly for persistent volumes, when scaling StatefulSets.

- Fix/Workaround:

• Increased the node pool size and resource limits for the StatefulSets.

• Rescheduled PVCs and balanced the resource requests more effectively across nodes.

- Lessons Learned:

StatefulSets require careful resource planning, especially for persistent storage.

- How to Avoid:

• Regularly monitor resource utilization, including storage, during scaling events.

• Ensure that node pools have enough capacity for StatefulSets and their associated storage requirements.

Scenario #488: Uncontrolled Resource Spikes After Scaling Large StatefulSets

- Category: Scaling & Load

- Environment: Kubernetes v1.22, GKE

- Summary: Scaling large StatefulSets led to resource spikes that caused system instability.

- What Happened:Scaling up a large StatefulSet resulted in CPU and memory spikes that overwhelmed the cluster, causing instability and outages.

- Diagnosis Steps:

  • Monitored CPU and memory usage and found that new StatefulSet pods were consuming more resources than anticipated.

  • Examined pod configurations and discovered they were not optimized for the available resources.

  Root Cause: Inefficient resource requests and limits for StatefulSet pods during scaling.

- Fix/Workaround:

  • Adjusted resource requests and limits for StatefulSet pods to better match the actual usage.

  • Implemented a rolling upgrade to distribute the scaling load more evenly.

- Lessons Learned:

  Always account for resource spikes and optimize requests for large StatefulSets.

- How to Avoid:

  • Set proper resource limits and requests for StatefulSets, especially during scaling events.

  • Test scaling for large StatefulSets in staging environments to evaluate resource impact.

Scenario #489: Cluster Autoscaler Preventing Scaling Due to Underutilized Nodes

- Category: Scaling & Load

- Environment: Kubernetes v1.24, AWS EKS

- Summary: The Cluster Autoscaler prevented scaling because nodes with low utilization were not being considered for scaling.

- What Happened:The Cluster Autoscaler was incorrectly preventing scaling because it did not consider nodes with low utilization, which were capable of hosting additional pods.

- Diagnosis Steps:

  • Reviewed Cluster Autoscaler logs and found that it was incorrectly marking low-usage nodes as “under-utilized” and therefore not scaling the cluster.

  • Observed that other parts of the cluster were under significant load