Kubernetes (K8S) Troubleshooting Knowledge Base

This space serves as the central knowledge base for diagnosing and resolving common issues within our Kubernetes environment. It is built from post-mortems and incident reports to create actionable, topic-based playbooks.

Categories:

- K8S Application Troubleshooting (Problems with workloads, e.g., Deployments, Pods, Services)
- (Problems with the control plane or cluster infrastructure, e.g., API Server, etcd)

K8S Application Troubleshooting

Guides for issues related to services running on the cluster.

1. Playbook: Troubleshooting Pod CrashLoopBackOff

Summary This state indicates that a pod's container is starting, crashing, and then being restarted by Kubernetes, in a continuous loop. The pod's status will show CrashLoopBackOff.

Common Root Causes

- **Missing Configuration:** The application crashes because a required environment variable, ConfigMap, or Secret is missing or incorrect.
- **Application Error:** The application itself is failing on startup (e.g., code panic, invalid configuration file, failed database connection).
- **Failed Liveness/Readiness Probes:** The probes are misconfigured (e.g., wrong port, bad health check endpoint, timeout too low), causing Kubernetes to kill the pod.
- **Incorrect Container command or args:** The entrypoint for the container is invalid or pointing to a non-existent file.

Diagnostic Playbook

1. **Check Pod Status:** Identify the failing pod.
2. **Describe the Pod:** This shows the *reason* for the current state and any events.

    - Look at the Events section for clues.
    - Look at the Last State section. It will often show Reason: Error and an Exit Code.
3. **Check Logs (Previous Container):** This is the **most critical step**. A pod in CrashLoopBackOff is constantly restarting, so its *current* logs are often empty. You must check the logs from the *previous* terminated container.
    - This log will almost always contain the fatal error that caused the crash (e.g., "Error: PAYMENT_API_KEY environment variable not set").

Resolution & Prevention

- **Resolution:**
    - Fix the underlying issue found in the logs (e.g., add the missing environment variable, correct the app code, fix the probe).
    - Apply the fix (e.g., kubectl apply -f deployment.yaml).
    - If necessary, trigger a rollout to pick up the new config: kubectl rollout restart deployment/<deployment-name> -n <namespace>.
- **Prevention:**
    - **Validate Configuration:** Use CI/CD linting tools (like kubeval or conftest) to ensure all required ConfigMaps, Secrets, and env vars are present in manifests before deployment.
    - **Application Fallbacks:** Code applications to handle missing non-critical configuration with sane defaults, rather than crashing.

Key Alert (Prometheus) Alert when any container is in a CrashLoopBackOff state.

Incident Example (K8S-APP-01)

- **Summary:** payment-service pods entered CrashLoopBackOff.
- **Root Cause:** A manual configuration update missed the PAYMENT_API_KEY environment variable, which was required by the application.
- **Resolution:** The missing env variable was added to the Deployment manifest, sourcing it from the payment-secrets Secret.

2. Playbook: Troubleshooting StatefulSet / Volume Attachment Failures

Summary This issue occurs when a pod (often part of a StatefulSet) fails to start and gets stuck in a Pending state. The pod's events show errors like FailedAttachVolume or FailedMount.

Common Root Causes

- **Cloud Provider Quotas:** You have hit a limit on the number of volumes that can be created in your account/region or attached to a single node.
- **Zone Mismatch:** The PersistentVolume (PV) exists in a different availability zone (e.g., us-east-1a) than the node the pod is scheduled on (e.g., us-east-1b). Cloud disks generally cannot be attached across zones.
- **StorageClass Issues:** The StorageClass specified in the PersistentVolumeClaim (PVC) does not exist or is misconfigured.
- **Volume in Use:** The volume is already mounted by another node (common with ReadWriteOnce (RWO) volumes).

Diagnostic Playbook

1. **Identify Pending Pod:**
2. **Describe the Pod:** This is the most important command. Check the Events section at the bottom.
    - Look for events like:
        - Warning FailedAttachVolume ... AttachVolume.Attach failed for volume "pvc-..." : ... cloud provider error: volume limit exceeded
        - Warning FailedMount ... MountVolume.NewVolume provisioner ... failed to provision volume ...
        - Warning FailedScheduling ... 0/3 nodes are available: 1 node(s) had a volume node affinity conflict.
3. **Check Cloud Provider Console:** Based on the error, log in to your cloud provider (AWS, GCP, Azure) and check:
    - Volume quotas (e.g., EBS volume limits per region, snapshot quotas).
    - Per-instance attachment limits.

Resolution & Prevention

- **Resolution:**
    - **Quota Issues:** Request an increase in your cloud provider volume limits.
    - **Zone Mismatch:** Ensure your StatefulSet/Deployment uses node affinity or that your StorageClass is configured with volumeBindingMode: WaitForFirstConsumer to delay volume creation until a pod is scheduled.
    - **Recreate Pod:** Once the underlying issue is fixed, you may need to delete the pod to force the scheduler to retry: kubectl delete pod <pod-name> -n <namespace>.
- **Prevention:**

    - **Monitor Quotas:** Implement alerts in your cloud provider to warn when you approach 80% of volume or attachment quotas.
    - **Pre-provision Headroom:** For critical StatefulSets, pre-provision a small buffer of PersistentVolumes.

Key Alert (Prometheus/Events) Alerting on FailedAttachVolume events is the most direct method (often requires an event-monitoring tool). You can also track the phase of PVs.

Incident Example (K8S-APP-02)

- **Summary:** orders-db StatefulSet failed to scale; new pods were stuck in Pending.
- **Root Cause:** kubectl describe pod revealed FailedAttachVolume events. The root cause was hitting the cloud provider's limit for total provisioned volumes in the region.
- **Resolution:** The limit was increased in the cloud provider console, and the pending pod was deleted to trigger a successful reschedule.

3. Playbook: Troubleshooting Application Latency & HPA Tuning

Summary Application APIs experience high P95/P99 latency (e.g., >1.5s) and user-facing timeouts, especially during traffic spikes. The Horizontal Pod Autoscaler (HPA) may not be reacting fast enough.

Common Root Causes

- **HPA Thresholds Too High:** The HPA's CPU or memory target (e.g., targetAverageUtilization: 90%) is too high. By the time the metric is breached, the application is already saturated and slow.
- **Slow Pod Startup:** The HPA scales up, but new pods take 1-2 minutes to start, pass readiness probes, and accept traffic, during which time the existing pods remain overwhelmed.
- **Scaling on the Wrong Metric:** The HPA is scaling on CPU, but the bottleneck is memory, I/O, or a downstream dependency (like a database).
- **Insufficient Resource requests:** Pod spec.resources.requests are set too low. The pod gets CPU-throttled *before* its utilization is high enough to trigger the HPA.

Diagnostic Playbook

1. **Check Metrics:** Use your observability platform (Grafana, Prometheus) to correlate API latency spikes with:
    - Deployment CPU/Memory utilization.
    - Number of pods (kube_deployment_spec_replicas).

    - Downstream dependencies (database query time, etc.).
2. **Check HPA Status:** See what the HPA is "thinking."
    - In this example, TARGETS shows 85%/70%. This means the *current* utilization is 85%, but the *target* is 70%. The HPA *should* be scaling up.
3. **Describe the HPA:** Check the Events section to see its recent scaling decisions.
    - If you see it scaling, but latency is still high, the issue is likely slow pod startup or the threshold being too high.

Resolution & Prevention

- **Resolution:**
    - **Immediate:** Manually scale the deployment to handle the load: kubectl scale deployment/<deployment-name> -n <namespace> --replicas=10.
    - **Long-term:** Tune the HPA. Edit the HPA (kubectl edit hpa <hpa-name>) to lower the target threshold (e.g., from 80% to 60-70%) to make it scale *sooner*.
- **Prevention:**
    - **Tune HPA Aggressively:** It's often better to scale up early (e.g., at 60% CPU) and scale down slowly.
    - **Use Custom Metrics:** If your app's load is better represented by requests_per_second (RPS) than CPU, use a custom Prometheus metric for the HPA.
    - **Pre-warming:** For predictable traffic (e.g., a planned sale), manually scale up the deployment *before* the traffic hits.

Key Alert (Prometheus) Alert directly on application latency, as this is the true user-facing symptom.

Incident Example (K8S-APP-03)

- **Summary:** user-service experienced >2s latency during peak traffic.
- **Root Cause:** The HPA was configured to scale only when CPU utilization hit 85%. This threshold was too high, and by the time scaling triggered, the app was already saturated.
- **Resolution:** The deployment was manually scaled, and the HPA target was permanently adjusted to 70%.

K8S Platform Troubleshooting

Guides for issues related to the cluster's core components (control plane).

1. Playbook: Troubleshooting API Server Throttling (429 Errors)

Summary The entire cluster feels slow. kubectl commands are sluggish or fail with 429 Too Many Requests. CI/CD pipelines fail to deploy, and pod/deployment updates are delayed.

Common Root Causes

- **Abusive Client / "Noisy Neighbor":** A single component (e.g., a CI/CD pipeline, a misconfigured monitoring agent, a custom controller) is spamming the API server with thousands of requests (especially expensive LIST or WATCH calls).
- **Controller Overload:** A large number of nodes, pods, and controllers all watching for changes can overwhelm the API server.
- **Insufficient API Server Resources:** The kube-apiserver pods themselves are under-provisioned for CPU or memory.

Diagnostic Playbook

1. **Check API Server Logs:** Look for 429 Too Many Requests entries.
2. **Check API Server Metrics (Prometheus):** This is the best way to find the source.
    - **Confirm Throttling:**
      A high or rapidly increasing value confirms throttling.
    - **Find the Culprit:** Use apiserver_request_total to find the "top talkers." Group by useragent, verb, and resource.

Resolution & Prevention

- **Resolution:**
    - **Immediate:** Identify and throttle the abusive client (e.g., pause the CI/CD pipeline, restart the failing controller).
    - **Short-term (Use with Caution):** If the load is legitimate, you can temporarily increase the API server's rate-limiting flags (e.g., --max-requests-inflight), but this is not a permanent fix.
- **Prevention:**
    - **Client-side Rate Limiting:** Ensure all custom automation (CI/CD, scripts) implements rate limiting and exponential backoff when interacting with the API server.

    - **Use Informers:** Custom controllers should use shared informers to cache resources locally instead of polling the API server with GET or LIST.
    - **Monitor 429s:** Alert on the apiserver_request_total{code="429"} metric.

Key Alert (Prometheus)

Incident Example (K8S-PLAT-01)

- **Summary:** Cluster-wide 429 errors from the API server.
- **Root Cause:** A CI/CD pipeline was configured with high concurrency, launching hundreds of parallel jobs that all polled the API server simultaneously.
- **Resolution:** CI/CD concurrency was reduced, and the pipeline was updated to use exponential backoff on API calls.

2. Playbook: Troubleshooting etcd Disk Latency

Summary The cluster control plane is sluggish, but it's not API server throttling. Pod scheduling is slow, API requests time out, and etcd logs show slow commits. This is a critical, cluster-endangering state.

Common Root Causes etcd is extremely sensitive to disk I/O latency.

- **I/O Contention:** (Most common) Other processes are competing for disk I/O on the etcd nodes. This includes:
    - Node-level logging agents (e.g., fluentd, filebeat).
    - Cluster backup jobs.
    - Security scanners.
- **Slow Disks:** The underlying storage is not fast enough (e.g., network-attached storage, standard spinning disks). etcd *requires* low-latency solid-state disks (SSDs).
- **"Noisy Neighbor":** In a public cloud, the underlying physical hardware is shared, and another customer's VM may be saturating the disk.

Diagnostic Playbook

1. **Check etcd Metrics (Prometheus):** This is the source of truth.
    - etcd_disk_backend_commit_duration_seconds: This measures how long it takes to commit a transaction to disk. The P99 (99th percentile) **must** be low (e.g., < 25ms). If this spikes, you have a disk I/O problem.
    - etcd_disk_wal_fsync_duration_seconds: Measures the write-ahead-log sync time. This should also be very low.
2.

3. **SSH to etcd Nodes:** If metrics are high, get on the nodes themselves.
    - iostat -x 1: Look for high %util (disk saturation), high await (I/O wait time), and a large avgqu-sz (queue size).
    - iotop: Identify which processes are consuming the most disk I/O. If you see etcd competing with fluentd, tar, or rsync, you've found the problem.

Resolution & Prevention

- **Resolution:**
    - **Immediate:** Immediately stop or throttle the competing process (e.g., systemctl stop fluentd, kill <backup-pid>). Cluster stability is the priority.
    - Stagger I/O-heavy jobs (like backups) so they don't run at the same time.
- **Prevention:**
    - **Isolate etcd:** This is the most important prevention.
        - Run etcd on dedicated nodes (control-plane nodes).
        - Do NOT run any other applications (especially logging agents or your own workloads) on these nodes.
        - Use the fastest possible disks (e.g., local NVMe SSDs, high-IOPS cloud disks).
    - **Schedule Jobs:** Schedule all backup, compaction, and scanning jobs for low-traffic, off-peak hours.
    - **Monitor Latency:** Aggressively alert on etcd commit duration.

Key Alert (Prometheus)

Incident Example (K8S-PLAT-02)

- **Summary:** High etcd disk latency (>50ms) caused slow pod scheduling.
- **Root Cause:** A cluster backup job and a log-shipping process ran at the same time, saturating the etcd nodes' disks.
- **Resolution:** The backup job was paused, and the schedule was changed to run 6 hours apart from the logging agent's heavy flushes.