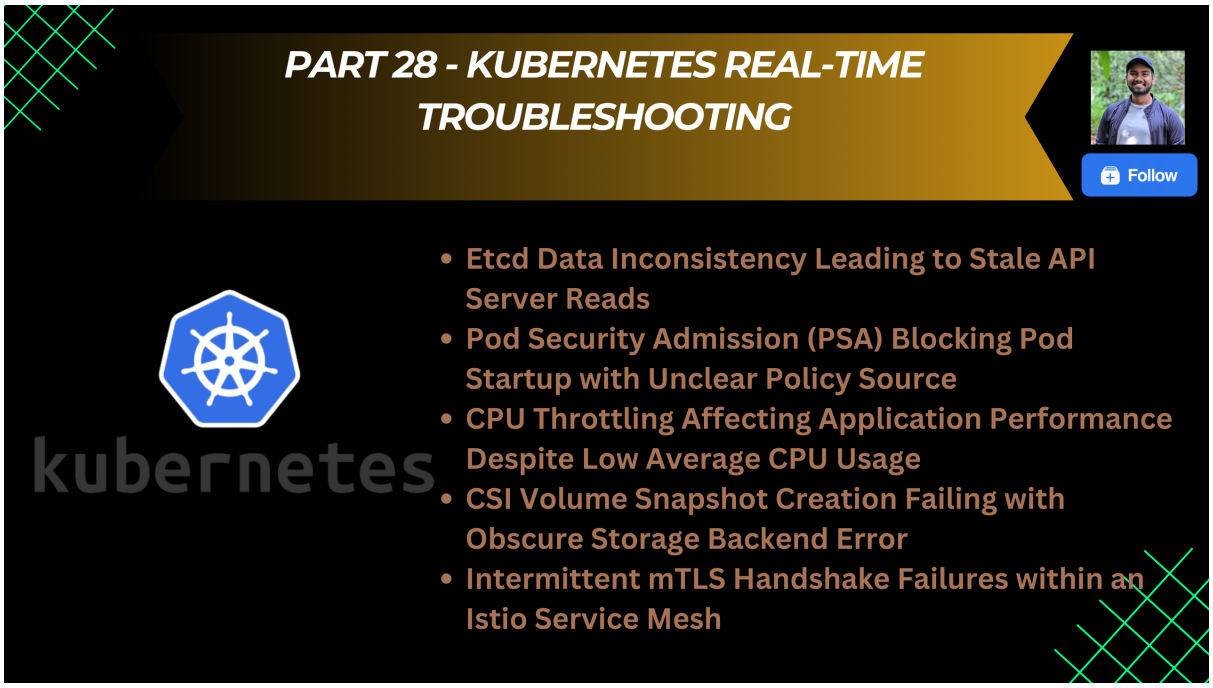
Part 28: Kubernetes Real-Time Troubleshooting

Introduction 🌐

Welcome to the world of Kubernetes troubleshooting, where every challenge is an opportunity to sharpen your skills and emerge victorious. Join us as we embark on a journey through common real-time scenarios, unraveling mysteries, and uncovering solutions along the way.



Scenario 136: Etcd Data Inconsistency Leading to Stale API Server Reads

Scenario

Users and controllers occasionally observe stale data when querying the Kubernetes API. For instance, a `kubectl get deployment my-app` might show an older `resourceVersion` or an outdated replica count, while a subsequent identical command a few seconds later shows the correct, updated information. The etcd cluster's high-level health check (`etcdctl endpoint health`) appears normal, but there's suspicion of subtle etcd instability affecting API server consistency.

Solution - Context:

kubectl config use-context k8s-c27-ha

Steps:

1. Confirm Stale Read Behavior:

Attempt to reproduce by rapidly updating a resource and immediately reading it back, possibly targeting different API server instances if load balanced.

Example: Update a ConfigMap and immediately get it

kubectl patch configmap my-config -n default --type merge -p '{"data":{"key":"newValue"'$(date +%s)""}}'

kubectl get configmap my-config -n default -o yaml | grep "key:"

Repeat quickly, or script this to hit the API multiple times

Note any discrepancies in `resourceVersion` or data returned.

2. Detailed Etcd Member Status and Performance:

On each etcd member node, run detailed status checks:

Replace ETCDCTL_ENDPOINTS, ETCDCTL_CACERT, ETCDCTL_CERT, ETCDCTL_KEY with your etcd client config

ETCDCTL_API=3 etcdctl --endpoints=$ETCDCTL_ENDPOINTS --cacert=$ETCDCTL_CACERT --cert=$ETCDCTL_CERT --key=$ETCDCTL_KEY \

endpoint status --write-out=json

Look for:

Variations in `raftIndex` or `raftTerm` across members (beyond normal lag).

Any member reporting `false` for `IsLeader` when it shouldn't be, or frequent leader elections.

High `DbSizeInUse` approaching quota limits.

3. Inspect Etcd Metrics:

If Prometheus is scraping etcd metrics, examine:

`etcd_server_leader_changes_seen_total`: Frequent leader changes are a bad sign.

`etcd_mvcc_db_total_size_in_bytes`: Monitor database size.

`etcd_disk_wal_fsync_duration_seconds_bucket`: High fsync latency indicates disk I/O problems.

`etcd_disk_backend_commit_duration_seconds_bucket`: High commit latency.

`etcd_network_peer_round_trip_time_seconds_bucket`: High RTT between peers.

Without Prometheus, you can curl the `/metrics` endpoint on each etcd member.

4. Examine Etcd Member Logs:

Check etcd logs on each member for warnings or errors related to:

Slow disk I/O (`wal_fsync_latency_too_long`, `backend_commit_latency_too_long`).

Network issues (`connection error`, `read tcp ...: i/o timeout`).

Leader elections (`elected leader`, `lost leader`).

Raft proposal failures or timeouts.

`mvcc: required revision is unavailable` or `mvcc: Revision ... is compacted`.

5. Check API Server Logs for Etcd-Related Errors:

On API server pods, look for errors indicating problems communicating with etcd or handling responses:

kubectl logs -n kube-system <apiserver-pod-name> | grep -i 'etcd\|stale\|too old resource version'

Messages like "too old resource version" can indicate an API server tried to write to etcd but its local cache was too far behind the current etcd revision, or that an API server is making a request to an etcd member that is significantly lagging.

6. Network Performance Between Etcd Members:

Ensure low latency and high bandwidth network connectivity between all etcd members. Use tools like `ping` and `iperf3` to test direct node-to-node network paths.

7. Resolve the Issue (Example: Problematic Etcd Member):

Assume metrics (e.g., high `wal_fsync_duration`) and logs point to one etcd member (`etcd-2`) having severe disk I/O issues.

Action:

1. Cordon and drain any workloads from the node hosting `etcd-2` (if it's not a dedicated etcd node).

2. Gracefully remove `etcd-2` from the cluster: `etcdctl member remove <etcd-2-member-id>`.

3. Diagnose and fix the disk issue on the node for `etcd-2` or reprovision the node.

4. Re-add the member (or a new one) to the etcd cluster: `etcdctl member add ...`.

5. Monitor cluster stability and consistency.

Outcome

The root cause of etcd inconsistency (e.g., a failing member, network partition between members, severe disk I/O contention) is identified and resolved.

Etcd members operate reliably, leader stability is restored, and commit latencies are low.

API servers provide consistent and timely reads of cluster state, eliminating stale data issues.

Scenario 137: Pod Security Admission (PSA) Blocking Pod Startup with Unclear Policy Source

Scenario

Pods in a newly configured namespace, `staging-apps`, are failing to start. `kubectl describe pod` shows the status `Pending` or `CreateContainerConfigError` with messages like: `"Error: failed to create pod: Pod "example-pod-xyz" is forbidden: violates PodSecurity "restricted:latest": non-default capabilities (NET_RAW), ..."`. The team believed they had set the `staging-apps` namespace to the `baseline` PSA profile, but pods are clearly being evaluated against `restricted`.

Solution - Context:

kubectl config use-context k8s-c28-secops

Steps:

1. Verify Pod Rejection Message:

   Deploy a test pod known to violate `restricted` but pass `baseline` into `staging-apps`:

```
# test-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: psa-test-pod
  namespace: staging-apps
spec:
  containers:
  - name: test-container
    image: busybox
    command: ["sh", "-c", "sleep 3600"]
    securityContext:
      capabilities:
        add: ["NET_ADMIN"] # Violates restricted, okay for baseline
```

Apply and describe:

```
kubectl apply -f test-pod.yaml
kubectl describe pod psa-test-pod -n staging-apps
```

Confirm the exact PSA violation message and the profile mentioned (e.g., `restricted:latest`).

2. Inspect Namespace Labels for PSA Configuration:

Check the labels on the `staging-apps` namespace:

```
kubectl get namespace staging-apps --show-labels
```

Look for PSA labels:

`pod-security.kubernetes.io/enforce`: Defines the active policy.

`pod-security.kubernetes.io/enforce-version`: (e.g., `latest`, `v1.28`)

`pod-security.kubernetes.io/audit`

`pod-security.kubernetes.io/warn`

Expected finding: The `enforce` label is likely missing, set to `restricted`, or an older version of the profile is causing unexpected behavior.

3. Check for Cluster-Wide Default PSA Configuration (AdmissionConfiguration):

The API server can be configured with a default PodSecurity admission plugin configuration. This applies if a namespace has no PSA labels.

Access the API server static pod manifest (usually `/etc/kubernetes/manifests/kube-apiserver.yaml` on control plane nodes).

Look for `--admission-control-config-file` and inspect that file. Within it, look for the `PodSecurity` plugin configuration:

```
# Example snippet from admission control config file
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1 # or v1beta1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "restricted" # This could be the source
      enforce-version: "latest"
      # ... audit, warn defaults ...
    exemptions:
      # ... any exempted namespaces/users ...
```

If a cluster-wide default like `restricted` is set here, it would apply to `staging-apps` if it lacks an `enforce` label.

4. Investigate ValidatingAdmissionPolicy (if applicable):

If your cluster uses `ValidatingAdmissionPolicy` (a more flexible CEL-based admission control mechanism, which can also enforce pod security rules), check for relevant policies:

```
kubectl get validatingadmissionpolicies
kubectl get validatingadmissionpolicybindings
```

Describe any policies that might be targeting pods in `staging-apps` or all namespaces, and examine their CEL expressions.

5. Resolve the Policy Conflict:

If Namespace Label is Incorrect/Missing: Apply the correct label to the namespace.

kubectl label namespace staging-apps pod-security.kubernetes.io/enforce=baseline pod-security.kubernetes.io/enforce-version=latest --overwrite

Also consider setting warn and audit labels

kubectl label namespace staging-apps pod-security.kubernetes.io/warn=baseline pod-security.kubernetes.io/warn-version=latest --overwrite

kubectl label namespace staging-apps pod-security.kubernetes.io/audit=restricted pod-security.kubernetes.io/audit-version=latest --overwrite

If Cluster-Wide Default is the Cause: The best practice is to use explicit namespace labels. If that's not immediately feasible and the cluster-wide default needs adjustment (with caution!), it requires modifying the API server's admission control configuration file and restarting API servers.

If ValidatingAdmissionPolicy is the Cause: Adjust or disable the problematic `ValidatingAdmissionPolicy` or its `ValidatingAdmissionPolicyBinding`.

6. Retry Pod Creation:

Delete the failed test pod and try creating it again:

kubectl delete pod psa-test-pod -n staging-apps

kubectl apply -f test-pod.yaml

The pod should now be created successfully if it complies with the `baseline` profile.

Outcome

The source of the unexpected `restricted` Pod Security Admission policy enforcement on the `staging-apps` namespace (e.g., missing/incorrect namespace label, overriding cluster-wide default, or a conflicting `ValidatingAdmissionPolicy`) is identified.

The PSA configuration is corrected, typically by applying the intended `baseline` profile labels to the namespace.

Pods compliant with the `baseline` profile can now be successfully scheduled and run in the `staging-apps` namespace.

Scenario 138: CPU Throttling Affecting Application Performance Despite Low Average CPU Usage

Scenario

A latency-sensitive application pod (`query-engine` in the `realtime-db` namespace) shows low average CPU utilization (e.g., 25-30% of its defined CPU limit) in `kubectl top pod`. However, the application experiences sporadic high p99 latencies, and internal application metrics indicate occasional task processing delays. The node hosting the pod is not under heavy CPU load.

Solution - Context:

kubectl config use-context k8s-c29-perf

Steps:

1. **Verify Application Latency and Low Average CPU:**

   Confirm application-level p99 latency spikes using application monitoring.

   Confirm low average CPU usage for the pod:

   kubectl top pod query-engine-xxxxxxxx-yyyyy -n realtime-db --containers

2. **Check for CPU Throttling Metrics:**

   The key is to look at instantaneous throttling, not just averages.

   Access cAdvisor metrics from the Kubelet on the node where the pod is running. You can typically port-forward to the Kubelet's HTTPS port (10250) or use a metrics collection agent like Prometheus.

   Metrics to check for the specific container:

   `container_cpu_cfs_throttled_periods_total`: The cumulative number of periods in which the container's CPU usage was throttled.

   `container_cpu_cfs_periods_total`: The cumulative number of enforcement periods.

   `container_cpu_usage_seconds_total`: Cumulative CPU time consumed.

   A consistently increasing `container_cpu_cfs_throttled_periods_total` indicates the container is being throttled. Calculate the throttling percentage: `(delta(throttled_periods) / delta(periods))  100`.

3. Inspect Pod CPU Requests and Limits:

Get the pod definition:

kubectl get pod query-engine-xxxxxxxx-yyyyy -n realtime-db -o yaml

Examine `spec.containers[].resources.requests.cpu` and `spec.containers[].resources.limits.cpu`.

Common Cause: The application might be very bursty. It uses low CPU on average but has short, intense bursts that hit the CPU limit, causing throttling. If `requests` are set much lower than `limits`, it falls into the `Burstable` QoS class.

4. Analyze Application's CPU Usage Pattern:

If possible, use application-specific profiling tools or more granular CPU monitoring to understand its micro-burst behavior.

Consider if the application is single-threaded or multi-threaded. A single very active thread can hit the limit of 1 CPU core even if the overall limit is higher (e.g., `limits.cpu: "2"` but one thread tries to use >1 core worth of processing for a short burst).

5. Review Node CPU Allocation and QoS:

Ensure the node has sufficient allocatable CPU.

Understand the pod's Quality of Service (QoS) class:

`Guaranteed`: If `requests.cpu == limits.cpu` (and memory also). Least likely to be throttled if within limits.

`Burstable`: If `requests.cpu < limits.cpu`. Can use more CPU up to limits if available, but can be throttled.

`BestEffort`: No requests/limits. Most likely to be throttled.

6. Implement and Test Solutions:

A) Increase CPU Limit: If the bursts are legitimate and necessary for performance, and the node has capacity, increase `limits.cpu`.

```
# In Deployment spec
resources:
  requests:
    cpu: "1" # Or current request
  limits:
```

cpu: "2" # Increased limit

B) Align Requests and Limits (Guaranteed QoS): For latency-sensitive apps, often best to set `requests.cpu == limits.cpu` to achieve `Guaranteed` QoS if the workload is predictable.

resources:

requests:

cpu: "1.5" # Matched

limits:

cpu: "1.5" # Matched

C) Optimize Application: If possible, optimize the application to smooth out CPU bursts or handle throttling more gracefully (e.g., internal queuing).

D) Use CPU Manager (Static Policy): On the node, if the Kubelet's CPU Manager policy is set to `static`, and the pod is `Guaranteed` QoS with integer CPU requests, it can get exclusive CPU cores, reducing interference and throttling likelihood. This is an advanced node-level configuration.

7. Monitor Throttling and Latency Post-Change:

After applying a change, closely monitor `container_cpu_cfs_throttled_periods_total` and application p99 latencies to confirm improvement.

Outcome

CPU throttling is identified as the cause of performance degradation despite low average CPU usage.

The pod's CPU requests/limits are adjusted (e.g., increasing limits, setting Guaranteed QoS), or the application is optimized.

CPU throttling is significantly reduced or eliminated, and application p99 latencies improve to acceptable levels.

Scenario 139: CSI Volume Snapshot Creation Failing with Obscure Storage Backend Error

Scenario

Attempts to create a `VolumeSnapshot` for a PersistentVolumeClaim (PVC) named `mysql-data-pvc` in the `prod-db` namespace are consistently failing. The `VolumeSnapshot` object's status shows `ReadyToUse: false` and an error condition with a message like `Failed to prepare snapshot: rpc error: code = Aborted desc = Snapshot operation failed on backend storage (err_code: 9007)`. The PVC is bound, and the MySQL pod using it is running correctly. The CSI driver is `com.example.storage-csi`.

Solution - Context:

kubectl config use-context k8s-c30-storage

Steps:

1. **Inspect VolumeSnapshot and VolumeSnapshotContent:**

   Get the `VolumeSnapshot` YAML and status:

   kubectl get volumesnapshot mysql-data-snapshot -n prod-db -o yaml

   Note the `status.error.message` and `status.boundVolumeSnapshotContentName`.

   If `boundVolumeSnapshotContentName` exists, get the `VolumeSnapshotContent` (VSC):

   kubectl get volumesnapshotcontent <vsc-name> -o yaml

   Check its `status.error.message` for potentially more details. If no VSC is created, the issue might be earlier in the process.

2. **Check Logs of CSI External-Snapshotter Sidecar:**

   The `csi-snapshotter` sidecar runs alongside the CSI driver's controller plugin. Find the CSI controller pod (often in `kube-system` or a driver-specific namespace).

   kubectl get pods -n <csi-driver-ns> -l app=<csi-controller-label>

   Examine logs of the `csi-snapshotter` container within that pod:

   kubectl logs -n <csi-driver-ns> <csi-controller-pod> -c csi-snapshotter -f

   Look for errors related to the specific `VolumeSnapshot` name or the underlying PVC/PV. These logs often translate the gRPC errors from the CSI driver.

3. Check Logs of the Main CSI Driver Controller Plugin:

In the same CSI controller pod, check logs of the main driver container (e.g., `com-example-storage-csi-plugin`):

kubectl logs -n <csi-driver-ns> <csi-controller-pod> -c <csi-driver-container-name> -f

These logs should show the actual gRPC calls (`CreateSnapshot`) to the storage backend and any responses or errors returned directly by the storage system's API via the driver. The "err_code: 9007" likely originates here.

4. Consult Storage Backend Documentation for Error Codes:

The error message "err_code: 9007" is specific to the storage backend (`com.example.storage`). Search the vendor's documentation for this error code to understand its meaning.

Example: "Error 9007" might mean "Snapshot feature not licensed," "Maximum snapshots per volume exceeded," "Insufficient snapshot reserve space," or "Volume is in a state that prohibits snapshots."

5. Verify CSI Driver and Snapshot CRD/Controller Versions:

Ensure the versions of the `VolumeSnapshot` CRDs, the `external-snapshotter` sidecar, and the CSI driver itself are compatible with each other and with the Kubernetes version. Mismatches can lead to unexpected errors.

6. Check Storage Backend Directly:

If you have access to the storage system's management interface (GUI/CLI):

Attempt to manually create a snapshot of the LUN/volume that backs the PV for `mysql-data-pvc`.

Check the storage system's event logs or alerts for errors related to snapshot operations.

Verify snapshot quotas, available snapshot space, and any specific volume configurations that might prevent snapshots.

7. Resolve the Issue (Example: Backend Quota Exceeded):

Assume the storage backend documentation reveals "err_code: 9007" means "Snapshot quota for this storage pool has been reached."

Action:

1. Log in to the storage backend system.
2. Delete old/unnecessary snapshots to free up quota.
3. Alternatively, increase the snapshot quota for the relevant storage pool or volume if possible.
4. Retry the `VolumeSnapshot` creation in Kubernetes. The CSI driver will attempt the operation again.

8. Monitor VolumeSnapshot Status:

After addressing the backend issue, watch the `VolumeSnapshot` status:

kubectl get volumesnapshot mysql-data-snapshot -n prod-db -w

It should eventually transition to `ReadyToUse: true` and the error condition should clear.

Outcome

The obscure storage backend error (e.g., "err_code: 9007") preventing CSI `VolumeSnapshot` creation is deciphered and linked to a specific condition on the storage system (like quota, licensing, or unsupported volume config).

The issue on the storage backend is rectified.

CSI `VolumeSnapshot` operations for the PVC now succeed, and snapshots become available for use.

Scenario 140: Intermittent mTLS Handshake Failures within an Istio Service Mesh

Scenario

In a cluster with Istio service mesh installed, applications in the `frontend-services` namespace intermittently fail to connect to services in the `backend-apis` namespace. Errors reported by `frontend` pods include "connection reset by peer," "upstream connect error or disconnect/reset before headers," or "SSL handshake failed." Istio's global `PeerAuthentication` is set to `STRICT` mTLS. Other service communications seem fine.

Solution - Context:

kubectl config use-context k8s-c31-istio

Steps:

1. **Confirm Intermittent Failure and Error Messages:**

   Tail logs of a `frontend-services` pod while trying to reproduce the issue:

   kubectl logs -n frontend-services <frontend-pod-name> -c <application-container> -f

   Note the exact error messages and timestamps.

2. **Examine Istio Proxy (Envoy) Logs:**

   On both a failing `frontend` pod and a target `backend-apis` pod, check the `istio-proxy` (Envoy) container logs. Increase Envoy log level if needed (`pilot-agent request POST /logging?level=debug`).

   kubectl logs -n frontend-services <frontend-pod-name> -c istio-proxy -f --tail=100

   kubectl logs -n backend-apis <backend-pod-name> -c istio-proxy -f --tail=100

   Look for:

   On the client-side (frontend): `upstream_reset_before_response_started{connection_termination}`, `TLS_error`, ` फाइव (UC)` (Upstream Connection failure), `UH` (No Healthy Upstream).

   On the server-side (backend): Messages about TLS handshake failures, certificate validation errors, or connection resets.

3. **Check Istio Configuration for mTLS:**

   Verify global `PeerAuthentication` in `istio-system`:

   kubectl get peerauthentication default -n istio-system -o yaml

   # Ensure spec.mtls.mode is STRICT

   Check for namespace-specific `PeerAuthentication` in `frontend-services` and `backend-apis` that might override the global default.

   Check `DestinationRule`s for services in `backend-apis` namespace. Ensure their `trafficPolicy.tls.mode` is set to `ISTIO_MUTUAL`.

   kubectl get destinationrules -n backend-apis -o yaml

4. **Inspect Envoy Secrets (Certificates):**

   Use `istioctl` to check the secrets (certificates) loaded by Envoy in the proxies:

For a frontend pod

istioctl proxy-config secret <frontend-pod-name>.frontend-services -o json | jq '.dynamicActiveSecrets[] | select(.name == "default") | .secret.tlsCertificate.certificateChain.inlineBytes' | base64 -d | openssl x509 -text -noout

For a backend pod

istioctl proxy-config secret <backend-pod-name>.backend-apis -o json | jq '.dynamicActiveSecrets[] | select(.name == "default") | .secret.tlsCertificate.certificateChain.inlineBytes' | base64 -d | openssl x509 -text -noout

Verify:

Certificates are present and not expired.

The `Issuer` and `Subject Alternative Name` (SAN) fields are correct (SAN should match the service account of the pod).

The certificate is signed by the correct Istio CA (check `istio-ca-root-cert` ConfigMap in `istio-system`).

5. Check Istiod (Pilot) and Citadel/Istio CA Logs:

Examine logs of `istiod` pods in `istio-system`:

kubectl logs -n istio-system -l app=istiod -f

Look for errors related to:

Certificate signing requests (CSRs) from proxies.

Pushing configuration (XDS) to proxies.

CA errors if Istiod is managing the CA. If using a separate Citadel, check its logs.

6. Verify Clock Skew:

Significant clock skew between nodes can cause TLS handshake failures due to certificate validity windows.

Check time on nodes hosting `frontend` and `backend` pods:

SSH to nodes

date

Ensure NTP is configured and synchronized.

7. Network Policies (Kubernetes Native):

Although Istio handles mTLS, ensure underlying Kubernetes NetworkPolicies are not inadvertently blocking traffic on port 15008 (Istio's mTLS port) or other ports Istio components use, which could disrupt XDS or certificate delivery.

8. Resolve the Issue (Example: Stale Certificates on Specific Pods):

Assume Envoy logs show certificate validation errors, and `istioctl proxy-config secret` reveals some pods have older, nearly expired, or mismatched certificates. This could be due to a past hiccup in `istiod`'s certificate rotation for those specific proxies.

Action:

1. Try restarting the affected `frontend` and/or `backend` pods one by one. This forces their Envoy proxies to request fresh certificates from `istiod`.

kubectl delete pod <problematic-pod-name> -n <namespace>

2. If the problem is widespread or persists after pod restarts, investigate `istiod` more deeply for issues with its CSR signing process or XDS delivery to specific proxies. Check `istiod` resource utilization.

Outcome

The root cause of intermittent mTLS handshake failures (e.g., stale/invalid certificates on specific proxies, `PeerAuthentication`/`DestinationRule` misconfiguration, clock skew, Istiod issue) is identified.

Corrective actions are taken (e.g., restarting pods to refresh certs, fixing Istio policies, synchronizing clocks, addressing Istiod problems).

Reliable mTLS-secured communication is restored between services in `frontend-services` and `backend-apis` namespaces.

In the up-coming parts, we will discussion on more troubleshooting steps for the different Kubernetes based scenarios. So, stay tuned for the and follow @Prasad Suman Mohan for more such posts.