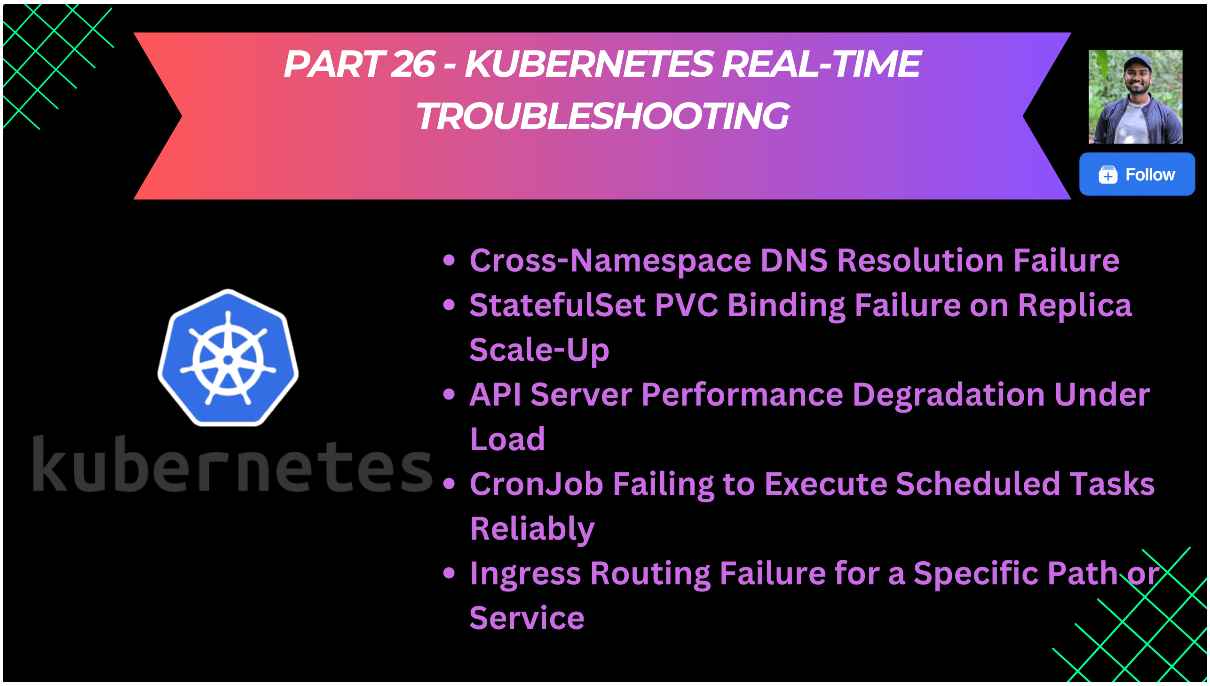
Part 26: Kubernetes Real-Time Troubleshooting

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 126: Cross-Namespace DNS Resolution Failure

Scenario:

Pods running in the `project-alpha` namespace are suddenly unable to resolve Services hosted within the `project-beta` namespace (e.g., `beta-service.project-beta.svc.cluster.local`). Internal DNS resolution for services within `project-alpha` works correctly, and pods in `project-beta` have no issues. This is disrupting inter-service communication vital for a multi-component application.

Solution

Context:

kubectl config use-context k8s-c17-prod

Steps:

1. **Verify DNS Resolution Failure:**

   Exec into a pod in `project-alpha`:

   kubectl exec -it -n project-alpha <alpha-pod-name> -- /bin/sh

   Attempt DNS lookups using `nslookup` or `dig` (install if needed: `apk add bind-tools` or `apt-get update && apt-get install dnsutils`):

   # Should fail or timeout

   nslookup beta-service.project-beta.svc.cluster.local

   # Should succeed

   nslookup alpha-service.project-alpha.svc.cluster.local

   # Should succeed

   nslookup kubernetes.default.svc.cluster.local

2. **Check CoreDNS Logs:**

   Find CoreDNS pods:

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

   Examine logs for errors related to `project-beta` lookups:

   kubectl logs -n kube-system <coredns-pod-name> -f

   Look for errors like `NXDOMAIN`, `SERVFAIL`, or specific plugin errors.

3. **Inspect Network Policies:**

   Check if any NetworkPolicy in `project-beta` restricts ingress from `project-alpha`:

   kubectl get networkpolicy -n project-beta

   Describe relevant policies to check `podSelector` and `ingress` rules:

   kubectl describe networkpolicy <policy-name> -n project-beta

Check if any NetworkPolicy in `kube-system` restricts egress from CoreDNS pods to the API server or if egress from `project-alpha` to `kube-system` (UDP/TCP 53) is blocked.

kubectl get networkpolicy -n kube-system

kubectl get networkpolicy -n project-alpha

4. **Verify Service and Endpoint Existence in `project-beta`:**

Ensure the target service exists and has endpoints:

kubectl get svc beta-service -n project-beta

kubectl get endpoints beta-service -n project-beta

If endpoints are missing, troubleshoot the pods backing `beta-service`.

5. **Review CoreDNS Configuration:**

Examine the CoreDNS ConfigMap:

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

Look for incorrect `forward` directives, plugin configurations (`kubernetes`, `rewrite`, `acl`), or potential misconfigurations affecting cross-namespace lookups. Ensure the `kubernetes` plugin configuration is standard.

6. **Test DNS from a Different Namespace:**

Exec into a pod in a third namespace (e.g., `default`) and attempt the lookup to rule out a `project-alpha`-specific issue:

kubectl exec -it -n default <default-pod-name> -- nslookup beta-service.project-beta.svc.cluster.local

7. **Implement Fix (Example: Network Policy Adjustment):**

If a NetworkPolicy in `project-beta` was blocking DNS lookups (CoreDNS needs to reach pods/endpoints), adjust it to allow ingress from `kube-system` namespace or specifically from CoreDNS pods:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-queries
```

```
  namespace: project-beta
spec:
  podSelector: {} # Apply to all pods in project-beta
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system # Or a more specific label for kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
    - protocol: UDP
      port: 53
    - protocol: TCP # Less common for queries, but good practice
      port: 53
  # ... include other necessary ingress rules here ...
```

Apply the updated policy:

```
kubectl apply -f allow-dns-policy.yaml -n project-beta
```

8. Re-Verify DNS Resolution:

Repeat Step 1 from the `project-alpha` pod.

Outcome:

Cross-namespace DNS resolution between `project-alpha` and `project-beta` is restored.

The root cause (e.g., overly restrictive NetworkPolicy, CoreDNS misconfiguration) is identified and rectified.

Inter-service communication dependent on DNS is functional again.

Scenario 127: StatefulSet PVC Binding Failure on Replica Scale-Up

Scenario

You scale up a StatefulSet named `event-processor` in the `project-stream` namespace from 3 to 5 replicas. The new pods (`event-processor-3` and `event-processor-4`) get stuck in the `Pending` state. Describing the pods reveals their PersistentVolumeClaims (PVCs) like `data-event-processor-3` cannot be bound. Existing replicas (`-0` to `-2`) are running fine with their PVCs bound.

Solution

Context:

kubectl config use-context k8s-c18-stage

Steps:

1. **Confirm Pod Status and Events:**

    Check pod status:

    kubectl get pods -n project-stream -l app=event-processor

    Describe a pending pod to see events related to scheduling and volume binding:

    kubectl describe pod event-processor-3 -n project-stream

    Look for events like `FailedScheduling` (if node resources are the issue) or `FailedBinding` (specific to PVC). Note the PVC name (`data-event-processor-3`).

2. **Inspect the Pending PersistentVolumeClaim (PVC):**

    Describe the specific PVC mentioned in the pod events:

    kubectl describe pvc data-event-processor-3 -n project-stream

    Look for events associated with the PVC itself. Common errors include:

    `no persistent volumes available for this claim`

`storageclass.storage.k8s.io "<storage-class-name>" not found`

Errors from the external provisioner (if using dynamic provisioning).

3. **Check Available PersistentVolumes (PVs):**

List PVs and filter by the relevant StorageClass:

kubectl get pv --sort-by=.metadata.creationTimestamp

kubectl get pv -l storageclass=<storage-class-name> # Get StorageClass from PVC description

Look for PVs in `Available` state that match the PVC's requirements (StorageClass, access modes, capacity).

Check if existing PVs are already bound to the older PVCs (`data-event-processor-0` to `-2`).

4. **Verify the StorageClass:**

Ensure the StorageClass specified in the PVC (or the default one) exists and is correctly configured:

kubectl get storageclass <storage-class-name> -o yaml

Check the `provisioner` field, `reclaimPolicy`, `volumeBindingMode` (`WaitForFirstConsumer` can delay binding until pod scheduling), and any `parameters`.

5. **Investigate the CSI Driver / External Provisioner:**

If using dynamic provisioning, check the logs of the CSI controller/external provisioner pods (often in `kube-system` or a dedicated namespace):

kubectl get pods -n <csi-driver-namespace>

kubectl logs -n <csi-driver-namespace> <csi-provisioner-pod-name> -f

Look for errors related to volume creation, API credentials, quota limits, or storage backend issues.

6. **Check Node-Specific Factors (if `volumeBindingMode: WaitForFirstConsumer`):**

If the StorageClass uses `WaitForFirstConsumer`, binding waits until a pod is scheduled. Check if scheduling is failing due to node resource constraints, taints/tolerations, or affinity rules:

kubectl describe pod event-processor-3 -n project-stream # Look for scheduling failures

kubectl get nodes # Check node conditions and capacity

Also check CSI node plugin logs on potential target nodes for attachment errors.

7. Resolve the Issue (Example: Insufficient Quota in Provisioner):

Assume CSI logs show errors like "Quota Exceeded" on the storage backend.

Action: Increase the storage quota on the backend system (e.g., AWS EBS, GCE PD, vSphere datastore).

Kubernetes doesn't need direct changes, but the provisioner should now succeed on its next retry.

8. Verify PVC Binding and Pod Startup:

Monitor the PVC status:

```
kubectl get pvc data-event-processor-3 -n project-stream -w
```

Once it changes to `Bound`, check the pod status:

```
kubectl get pods event-processor-3 -n project-stream -w
```

The pod should transition from `Pending` to `ContainerCreating` and then `Running`. Repeat for `event-processor-4`.

Outcome:

The root cause of the PVC binding failure (e.g., lack of available PVs, StorageClass issue, provisioner error, quota limit) is identified and resolved.

The new StatefulSet replicas (`event-processor-3`, `event-processor-4`) successfully bind their PVCs.

The pods start correctly, and the StatefulSet reaches the desired scale of 5 replicas.

<u>Scenario 128: API Server Performance Degradation Under Load</u>

Scenario

Users report intermittent slowness and timeouts when using `kubectl`. CI/CD pipelines interacting with the cluster are failing sporadically. Monitoring reveals high latency and increased CPU/Memory usage for `kube-apiserver` pods, but certificates are valid, and etcd seems healthy (`etcdctl endpoint health` is OK). The cluster isn't throwing obvious errors, just becoming sluggish.

Solution

Context:

kubectl config use-context k8s-c19-perf

Steps:

1. **Confirm API Server Slowness:**

Run simple, repeated `kubectl` commands and measure response time:

time kubectl get nodes

time kubectl get pods -A -l app=some-common-label --limit 10

Compare response times during peak vs. off-peak hours if possible.

2. **Monitor API Server Resources and Metrics:**

Check resource utilization of API server pods:

kubectl top pod -n kube-system -l component=kube-apiserver

Use Prometheus/Grafana (or similar) to examine key API server metrics:

`apiserver_request_duration_seconds_bucket`: High latency distribution?

`apiserver_request_total`: High overall request rate? Filter by `verb`, `resource`, `client`.

`apiserver_current_inflight_requests`: High number of concurrent requests?

`workqueue_depth`: For controllers, indicates potential processing delays.

`etcd_request_duration_seconds_bucket`: Rule out etcd latency being the primary cause (even if health is okay).

3. **Identify Heavy API Clients/Controllers:**

Analyze `apiserver_request_total` broken down by `client` (User Agent) and `resource`/`verb`.

Look for specific clients (e.g., a custom controller, monitoring agent, CI/CD system) making excessive `LIST` or `WATCH` calls, especially on large resource sets (Pods, Events, ConfigMaps).

Check API server audit logs for frequent requests from specific IPs or service accounts if enabled.

4. Inspect API Server Logs:

Check logs for throttling messages or specific errors:

```
kubectl logs -n kube-system <kube-apiserver-pod-name> | grep -i
'throttling\|timeout\|error'
```

5. Review Resource Limits/Requests for API Server:

Check the manifest used to deploy the API server (often a static pod manifest in `/etc/kubernetes/manifests/kube-apiserver.yaml` on control plane nodes):

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep -A2 resources:
```

Are the CPU/memory requests and limits adequate for the observed load?

6. Consider API Priority and Fairness (APF):

Check if APF is enabled (default in recent versions):

```
kubectl get flowschemas
```

```
kubectl get prioritylevelconfigurations
```

APF might be throttling less important requests to protect the server, which could be perceived as slowness by those clients. Check metrics related to APF (e.g., `apiserver_flowcontrol_request_wait_duration_seconds`).

7. Optimize Problematic Clients:

If a specific client/controller is identified as the source of excessive load:

Reduce LIST/WATCH scope: Use label selectors, field selectors, or watch specific objects instead of entire collections.

Increase sync intervals: Don't reconcile resources too frequently if not needed.

Use Informers/Caches: Ensure controllers use shared informers to cache objects locally instead of querying the API server repeatedly.

Implement client-side throttling: Limit the rate of requests the client makes.

8. Scale Up/Tune API Server (If Necessary):

If the overall legitimate load is high, consider:

Increasing CPU/Memory limits and requests for the API server pods.

Scaling out the number of API server replicas (requires load balancing).

Tuning API server flags (e.g., `--max-requests-inflight`, `--max-mutating-requests-inflight`), use with caution.

9. Monitor Post-Changes:

Continuously monitor API server metrics (latency, resource usage) after applying fixes (client optimization or server scaling).

Verify `kubectl` responsiveness and CI/CD pipeline stability.

Outcome

The bottleneck causing API server slowness is identified (e.g., a misbehaving controller, insufficient resources, high legitimate load).

Corrective actions are taken, such as optimizing API clients or scaling/tuning the API server.

API server performance and responsiveness are restored to acceptable levels.

Cluster interaction stability (kubectl, CI/CD) is improved.

<u>Scenario 129: CronJob Failing to Execute Scheduled Tasks Reliably</u>

Scenario

A critical nightly CronJob named `daily-report` in the `reporting` namespace, scheduled to run at `0 2   ` (2 AM daily), has become unreliable. Sometimes it runs, creating a Job pod, but other times it completely misses its schedule, and no Job is created. There are no obvious errors in the CronJob controller logs.

Solution

Context:

kubectl config use-context k8s-c20-batch

Steps:

1. **Verify CronJob Definition and Status:**

Get the full CronJob definition:

kubectl get cronjob daily-report -n reporting -o yaml

Check:

`schedule`: Is it `0 2   `? Verify syntax correctness.

`suspend`: Is it `false` or unset? If `true`, the CronJob is paused.

`concurrencyPolicy`: (`Allow`, `Forbid`, `Replace`) - Could `Forbid` prevent runs if a previous job is stuck?

`successfulJobsHistoryLimit` / `failedJobsHistoryLimit`: Are old jobs being cleaned up?

`startingDeadlineSeconds`: Is it set? If too short, a missed schedule due to controller lag might cause the job to be skipped entirely.

2. **Check Recent Job History:**

List Jobs created by this CronJob:

kubectl get jobs -n reporting -l cronjob=daily-report --sort-by=.metadata.creationTimestamp

Note the timestamps of the created Jobs. Are there gaps corresponding to the missed schedules?

Check the status of recent Jobs (Succeeded, Failed). A failed job might impact future runs depending on `concurrencyPolicy`.

3. **Examine Kubernetes Events:**

Events often provide clues about scheduling decisions or failures:

kubectl get events -n reporting --field-selector involvedObject.kind=CronJob,involvedObject.name=daily-report

kubectl get events -n reporting --field-selector involvedObject.kind=Job # Look for creation/deletion events

Look for events like `SuccessfulCreate`, `FailedCreate`, or potentially scheduling issues related to the Job's pod template.

4. **Check Controller Manager Logs:**

The CronJob controller runs within the `kube-controller-manager`. Check its logs on the master node(s):

Find the pod

kubectl get pods -n kube-system -l component=kube-controller-manager

Check logs, grep for the CronJob name or namespace

kubectl logs -n kube-system <kube-controller-manager-pod-name> | grep 'daily-report\|reporting'

Look for errors related to creating Jobs, time synchronization issues, or controller loops being slow.

5. Investigate Resource Quotas:

Does the `reporting` namespace have ResourceQuotas that might prevent Job or Pod creation?

kubectl get resourcequota -n reporting

kubectl describe resourcequota <quota-name> -n reporting

Check if limits for `jobs.batch`, `pods`, `cpu`, or `memory` are being hit around 2 AM.

6. Verify RBAC Permissions for the Job's ServiceAccount:

Identify the ServiceAccount used by the Job template within the CronJob spec (`spec.jobTemplate.spec.template.spec.serviceAccountName`).

Ensure this ServiceAccount has the necessary permissions (e.g., create Pods, access Secrets/ConfigMaps) required by the Job's container. While this usually causes Job failure rather than non-creation, it's worth checking if related events are confusing.

kubectl describe rolebinding,clusterrolebinding -n reporting | grep <serviceAccountName>

Or use a tool like rbac-lookup

7. Consider Time Synchronization:

Ensure all nodes (especially control plane nodes running `kube-controller-manager`) have accurate and synchronized time (e.g., using NTP). Significant time skew can disrupt schedules.

Check time on master node: `date`

8. Address the Likely Cause (Example: `startingDeadlineSeconds`):

Assume investigation suggests the controller manager was occasionally busy or delayed around 2 AM, and the default or a very short `startingDeadlineSeconds` caused the CronJob controller to skip the run if it couldn't create the Job very close to the scheduled time.

Solution: Add or increase `startingDeadlineSeconds` in the CronJob spec to allow more flexibility:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-report
  namespace: reporting
spec:
  schedule: "0 2   "
  startingDeadlineSeconds: 120 # Allow 2 minutes past schedule before skipping
  jobTemplate:
  # ... rest of template
```

Apply the change:

```
kubectl apply -f daily-report-cronjob.yaml
```

9. Monitor Over Several Cycles:

Observe the CronJob over the next few days, checking Job history (Step 2) to confirm it now runs reliably at 2 AM.

Outcome:

The reason for the CronJob's unreliable execution (e.g., missed schedule deadline, resource quota exhaustion, controller lag) is identified.

The CronJob configuration is adjusted (e.g., adding `startingDeadlineSeconds`, clearing stuck jobs, increasing quotas).

The `daily-report` CronJob now reliably creates its associated Job at the scheduled time.

Scenario 130: Ingress Routing Failure for a Specific Path or Service

Scenario

An Ingress resource is configured to route traffic to multiple backend services. Requests to `https://app.example.com/serviceA` work correctly, mapping to `service-a` in the `apps` namespace. However, requests to `https://app.example.com/serviceB` consistently return a 503 Service Unavailable or 404 Not Found error from the Ingress controller, even though `service-b` and its pods in the `apps` namespace appear healthy.

Solution

Context:

kubectl config use-context k8s-c21-gateway

Steps:

1. **Verify Pod and Service Health for `service-b`:**

   Check pods backing `service-b`:

   kubectl get pods -n apps -l app=service-b -o wide

   Ensure they are `Running` and `Ready`. Check their logs for internal errors.

   Check the Service definition:

   kubectl get svc service-b -n apps -o yaml

   Verify `selector`, `port`, and `targetPort`.

   Check the Endpoints object associated with the service:

   kubectl get endpoints service-b -n apps

   Ensure it lists the correct IP addresses and ports of the healthy `service-b` pods. If empty or incorrect, troubleshoot the pod selectors or pod readiness.

2. **Inspect the Ingress Resource Definition:**

   Get the full Ingress definition:

   kubectl get ingress main-ingress -n apps -o yaml

   Carefully review the `rules` and `paths` sections:

   Is there a rule for `host: app.example.com`?

Under that host, is there a `path` entry for `/serviceB`?

Is the `pathType` correct (`Prefix`, `Exact`, `ImplementationSpecific`)? Mismatches are common causes. (`Prefix` is often safest).

Does the `backend` for `/serviceB` correctly specify `service.name: service-b` and `service.port.name` (or `number`) matching the Service definition? Check for typos.

Are there any relevant annotations specific to your Ingress controller (e.g., rewrite rules, backend protocol) that might be misconfigured for `/serviceB`?

3. Check Ingress Controller Logs:

Find the Ingress controller pods (e.g., `nginx-ingress`, `traefik`, `haproxy-ingress` - often in a dedicated namespace like `ingress-nginx` or `kube-system`):

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

Tail the logs while sending a request to `https://app.example.com/serviceB`:

```
kubectl logs -n <ingress-controller-namespace> <ingress-controller-pod-name> -f
```

Look for log entries corresponding to the failed request. They often indicate:

Which upstream (service endpoint) it tried to connect to.

Connection errors (`connect() failed`, `timeout`).

Configuration errors (`could not find backend`, `rule syntax error`).

HTTP status codes returned from the backend service (if the connection succeeded but the app failed).

4. Verify Network Connectivity from Ingress Controller to Pods:

If logs suggest connection issues:

Check Network Policies in the `apps` namespace that might block ingress from the Ingress controller's namespace/pods.

```
kubectl get networkpolicy -n apps
```

Exec into the Ingress controller pod and try to reach `service-b`'s ClusterIP or pod IPs directly (using `curl`, `wget`):

```
kubectl exec -it -n <ingress-controller-namespace> <ingress-controller-pod-name> -- /bin/sh

curl -v http://<service-b-cluster-ip>:<service-port>/

curl -v http://<service-b-pod-ip>:<target-port>/ # Try a known pod IP
```

5. Examine Ingress Controller Configuration:

Some controllers generate internal configuration (e.g., `nginx.conf` for nginx-ingress). If possible, inspect this generated config inside the controller pod to see how it interpreted the Ingress resource.

```
# Example for nginx-ingress
kubectl exec -it -n <ingress-ns> <nginx-pod> -- cat /etc/nginx/nginx.conf
```

Look for the server block corresponding to `app.example.com` and the location block for `/serviceB`.

6. Address the Likely Cause (Example: Incorrect `pathType`):

Assume the Ingress definition used `pathType: Exact` for `/serviceB`, but requests were coming in as `/serviceB/` (with a trailing slash). `Exact` matching would fail.

Solution: Change `pathType` to `Prefix` in the Ingress definition:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: main-ingress
  namespace: apps
spec:
  rules:
  - host: app.example.com
    http:
      paths:
      - path: /serviceA
        pathType: Prefix # Assuming this was already Prefix
        backend:
          service:
            name: service-a
            port:
              number: 80
      - path: /serviceB # Path that was failing
```

```yaml
          pathType: Prefix # Changed from Exact
          backend:
            service:
              name: service-b
              port:
                number: 8080
  # ... tls section ...
```

Apply the change:

```
kubectl apply -f main-ingress.yaml
```

7. Test the Endpoint Again:

Send a request to `https://app.example.com/serviceB` (and maybe `https://app.example.com/serviceB/`):

```
curl -k https://app.example.com/serviceB # Use -k if using self-signed certs
```

Expect a successful response (e.g., 200 OK) from `service-b`.

Outcome:

The specific misconfiguration preventing Ingress routing to `service-b` (e.g., incorrect `path`, `pathType`, service name/port, NetworkPolicy block, backend pod issue) is found and corrected.

Requests to `https://app.example.com/serviceB` are now successfully routed by the Ingress controller to the healthy `service-b` pods.

Full application functionality dependent on both `/serviceA` and `/serviceB` paths is restored.

Stay tuned for Part 27, where we will continue navigating the complexities of Kubernetes troubleshooting! Follow for more insights.

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.