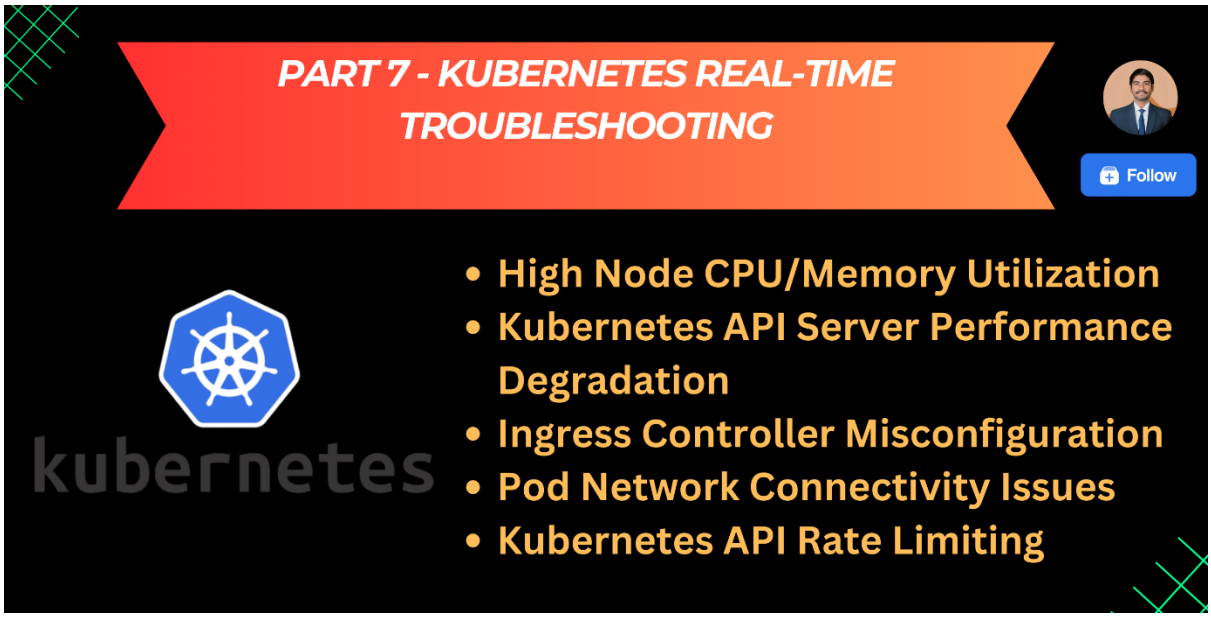
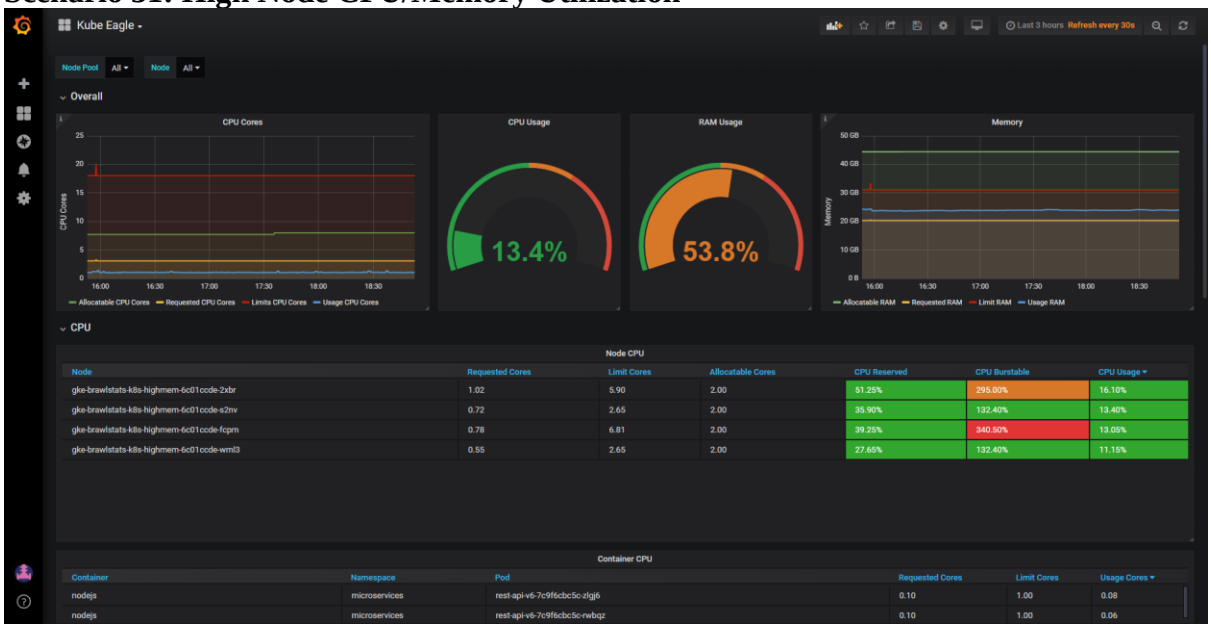
Part 7 - Kubernetes Real-Time Troubleshooting

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 31: High Node CPU/Memory Utilization



FOLLOW – Prasad Suman Mohan (for more updates)

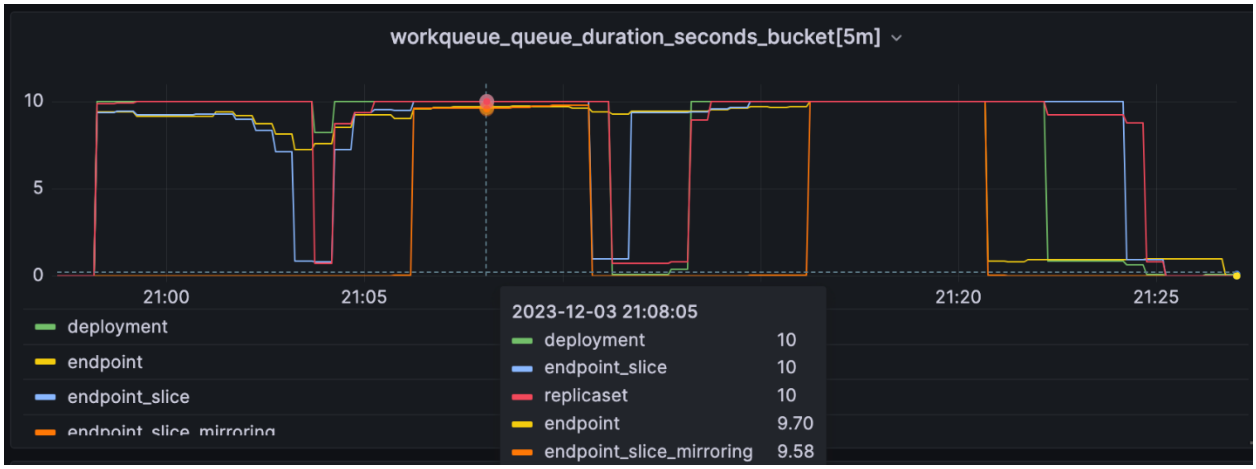
Symptoms: Kubernetes nodes experience high CPU or memory utilization, impacting pod scheduling and resource allocation.

Diagnosis: Monitor node resource metrics (`kubectl top node`) and review node logs for any kernel or system-level errors affecting resource utilization.

Solution:

1. Identify and terminate resource-intensive processes or containers consuming excessive CPU or memory resources on Kubernetes nodes.
2. Implement horizontal pod autoscaling (HPA) to automatically scale pod replicas based on resource usage metrics and demand patterns.
3. Optimize pod resource requests and limits to ensure efficient resource utilization and prevent resource contention between pods.
4. Scale out the Kubernetes cluster by adding additional nodes or upgrading existing node hardware to accommodate increased resource demands and workload growth.

Scenario 32: Kubernetes API Server Performance Degradation



Symptoms: Kubernetes API server becomes unresponsive or experiences slow response times, impacting cluster management operations.

Diagnosis: Monitor Kubernetes API server metrics (e.g., latency, throughput) and review API server logs for any errors or performance bottlenecks.

Solution:

1. Scale Kubernetes API server horizontally by deploying multiple replicas behind a load balancer to distribute incoming requests and improve availability.
2. Optimize etcd performance by tuning etcd configuration parameters (e.g., storage backend, snapshotting intervals) and upgrading etcd cluster hardware for better performance.
3. Implement caching mechanisms (e.g., kube-apiserver caching, client-side caching) to reduce API server load and improve response times for frequently accessed resources.
4. Monitor and tune API server resource utilization (e.g., CPU, memory, network) to ensure optimal performance and prevent resource exhaustion during peak loads.

Scenario 33: Ingress Controller Misconfiguration

```
apiVersion: v1
kind: secrets
metadata:
  name: secret-tls-test
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tls-example
spec:
  tls:
  - hosts:
    - ssltest.example.com
    secretName: secret-tls-test
  rules:
    - host: ssltest.example.com
      http:
        paths:
        - path: /
          backend:
            serviceName: serv1
            servicePort: 80
```

Symptoms: Ingress resources fail to route incoming traffic to backend services, resulting in HTTP 404 or connection refused errors for external clients.

Diagnosis: Review Ingress resource definitions (`kubectl get ingress`) and inspect ingress controller logs for any configuration errors or routing failures.

Solution:

1. Validate Ingress resource annotations and backend service endpoints to ensure correct routing rules and path mappings for incoming requests.
2. Verify DNS resolution for hostnames specified in Ingress rules and ensure that external DNS records point to the correct load balancer or Ingress controller IP address.
3. Check ingress controller configuration (e.g., Nginx, HAProxy) for any misconfigurations or limitations that may affect traffic routing and request handling.
4. Monitor network traffic and ingress controller metrics to identify any anomalies or performance issues affecting traffic throughput and latency.

Scenario 34: Pod Network Connectivity Issues

```
c:\>pathping 8.8.8.8

Tracing route to google-public-dns-a.google.com [8.8.8.8]
over a maximum of 30 hops:
  0  IAD743386 [199.38.148.54]
  1  199.38.148.53
  2  ix-xe-4-3-4-0.tcore1.AEQ-Ashburn.as6453.net [66.198.155.49]
  3  66.198.154.14
  4  108.170.246.65
  5  216.239.59.65
  6  google-public-dns-a.google.com [8.8.8.8]

Computing statistics for 150 seconds...
            Source to Here   This Node/Link
Hop  RTT    Lost/Sent = Pct  Lost/Sent = Pct  Address
  0                                           IAD743386 [199.38.148.54]
                                    0/ 100 =  0%   |
  1    0ms     0/ 100 =  0%     0/ 100 =  0%  199.38.148.53
                                    0/ 100 =  0%   |
  2    0ms     0/ 100 =  0%     0/ 100 =  0%  ix-xe-4-3-4-0.tcore1.AEQ-Ashburn.as6453.net [66.198.155.49]
                                    0/ 100 =  0%   |
  3    0ms     0/ 100 =  0%     0/ 100 =  0%  66.198.154.14
                                    0/ 100 =  0%   |
  4    1ms     0/ 100 =  0%     0/ 100 =  0%  108.170.246.65
                                    0/ 100 =  0%   |
  5   ---     100/ 100 =100%   100/ 100 =100%  216.239.59.65
                                    0/ 100 =  0%   |
  6    0ms     0/ 100 =  0%     0/ 100 =  0%  google-public-dns-a.google.com [8.8.8.8]

Trace complete.
```

Symptoms: Pods experience intermittent network connectivity issues, such as packet loss, latency spikes, or DNS resolution failures, impacting communication with other pods or external services.

Diagnosis: Use network troubleshooting tools (e.g., ping, traceroute, nslookup) inside pods to diagnose network connectivity problems and check network plugin logs for any errors or configuration issues.

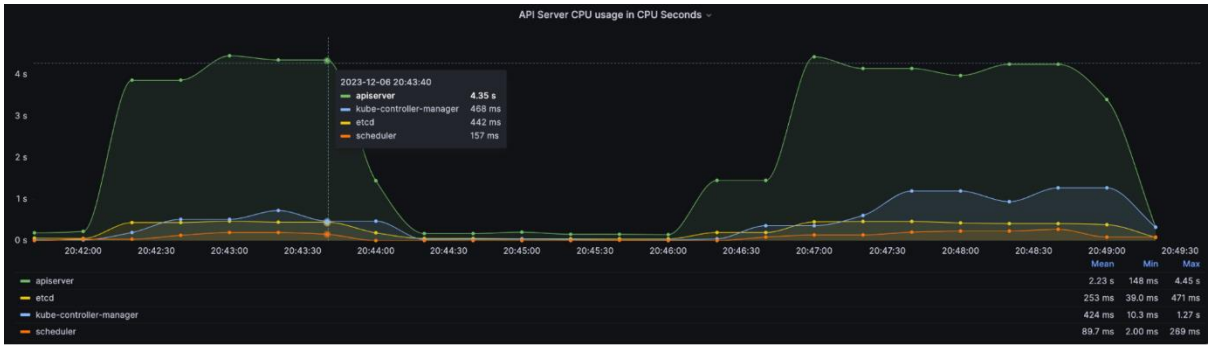
Solution:

1. Verify pod network configuration (e.g., CNI plugin, network policies) and ensure that pods have proper network connectivity to other pods within the cluster and external services outside the cluster.
2. Check for network interface misconfigurations (e.g., MTU settings, IP addressing, subnet overlaps) that may cause network packet drops or routing errors.
3. Review firewall rules and network security policies (e.g., AWS Security Groups, GCP Firewall Rules) to allow inbound and outbound traffic for pod communication.
4. Monitor pod network performance metrics (e.g., bandwidth, throughput, latency) and analyze network traffic patterns to identify and mitigate potential bottlenecks or congestion points.

Scenario 35: Kubernetes API Rate Limiting

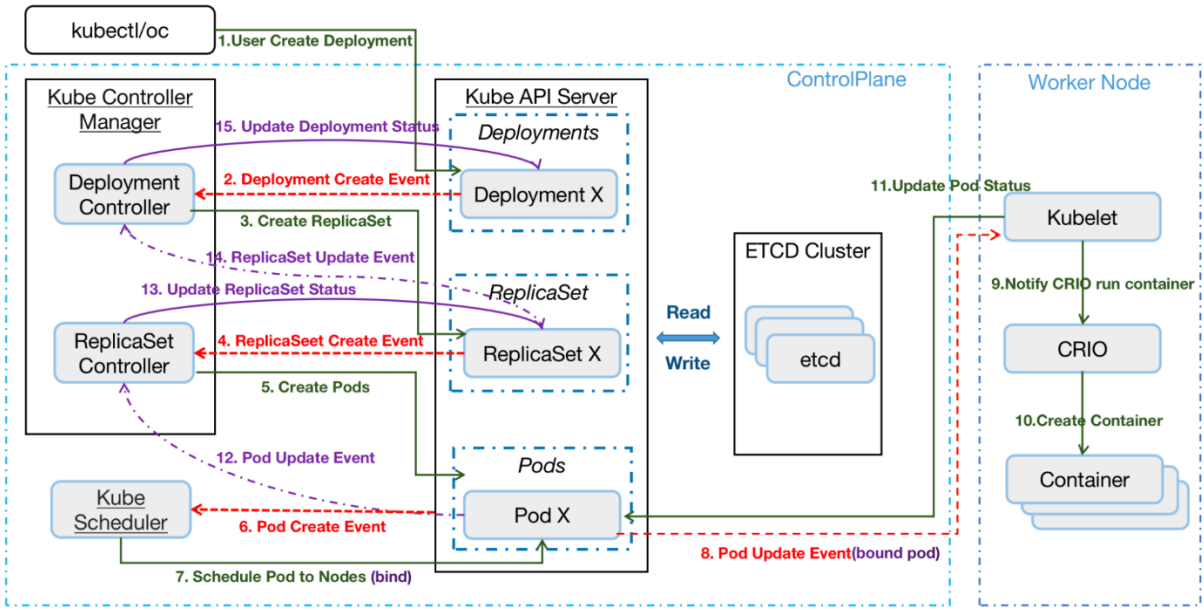
Symptoms: Kubernetes API requests are rate-limited or rejected due to exceeding API rate limits, causing delays or failures in cluster management operations.

Diagnosis: Monitor Kubernetes API server metrics (e.g., request rate, latency, errors) and review audit logs (`kubectl logs -n kube-system kube-apiserver`) for any indications of rate-limiting enforcement or API throttling events.



Solution:

1. Adjust Kubernetes API rate limits (e.g., `--max-requests-inflight`, `--max-mutating-requests-per-second`, `--max-requests-per-second`) to accommodate higher request volumes and prevent API throttling during peak loads.
2. Implement API request batching or aggregation techniques to reduce the number of individual requests and optimize API server performance under heavy request loads.
3. Scale Kubernetes API server horizontally by deploying multiple replicas behind a load balancer and distributing incoming requests evenly across API server instances to handle higher request throughput.
4. Optimize client-side API usage and avoid unnecessary or redundant API calls by batching requests, caching responses, and minimizing polling intervals to reduce overall API traffic and alleviate rate-limiting constraints.



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.

FOLLOW – Prasad Suman Mohan (for more updates)