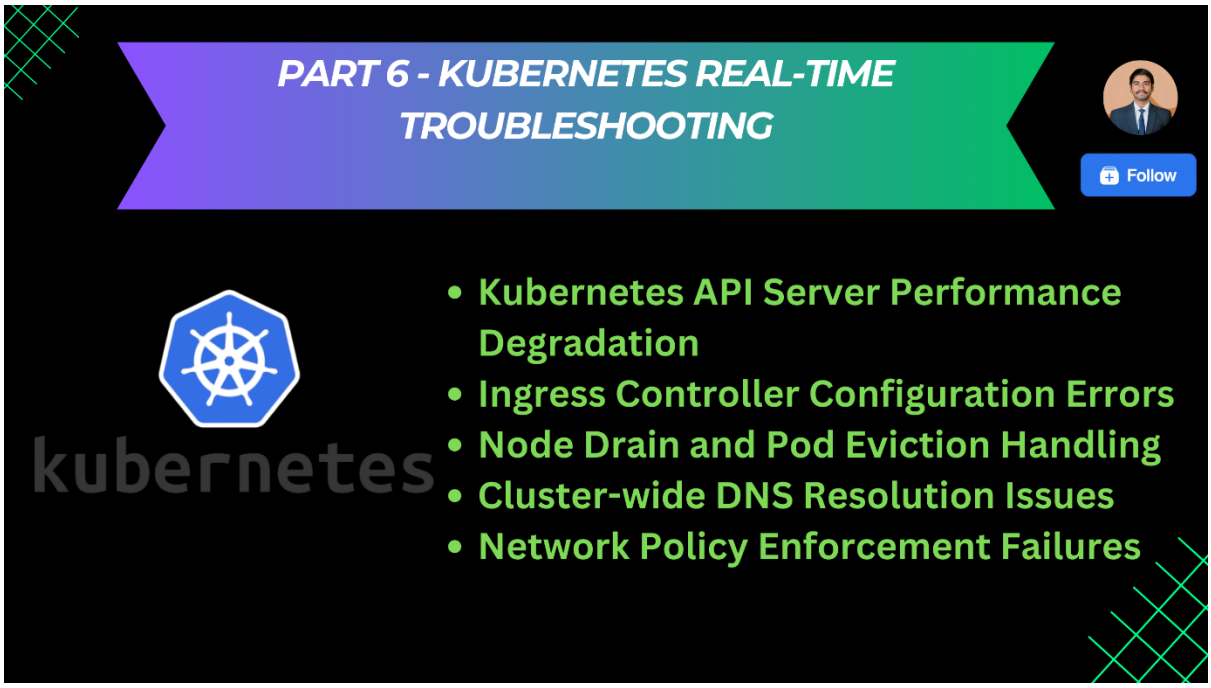
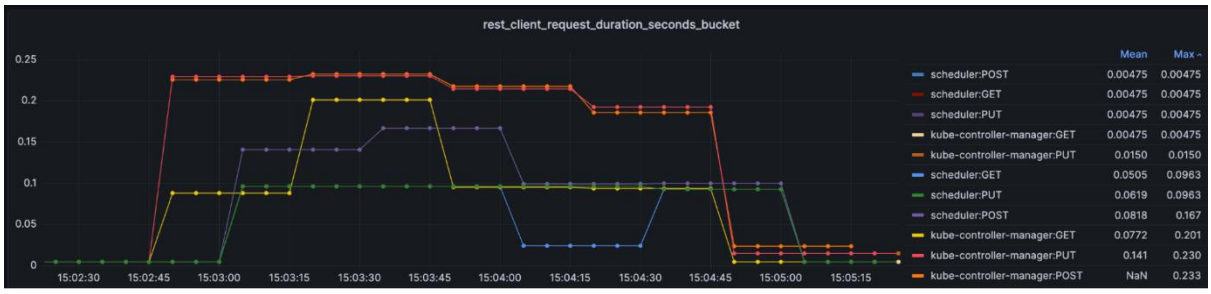
Part 6 - Kubernetes Real-Time Troubleshooting

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 26: Kubernetes API Server Performance Degradation



Symptoms: Kubernetes API server response times increase or become unresponsive, leading to delays in API operations and cluster management tasks.

Diagnosis: Monitor Kubernetes API server metrics (e.g., latency, throughput) using monitoring tools like Prometheus and Grafana.

FOLLOW – Prasad Suman Mohan (for more updates)

Solution:

1. Scale Kubernetes API server replicas horizontally to distribute incoming requests and improve response times.
2. Optimize etcd performance by tuning storage backend parameters and ensuring hardware resources (CPU, memory, disk) meet recommended specifications.
3. Review and optimize Kubernetes API server configuration flags (e.g., --max-requests-inflight, --max-mutating-requests) to prevent resource contention and improve concurrency handling.
4. Implement API request throttling or rate limiting mechanisms to prevent overload and protect API server performance during peak usage periods.

Scenario 27: Ingress Controller Configuration Errors

```
E0629 19:54:09.340338       8 queue.go:130] "requeuing" err=<
        ----------------------------------------------------------------------------
        Error: exit status 1
        2022/06/29 19:54:09 [warn] 179#179: the "http2_max_field_size" directive is obsolete, use the "large_client_header_bu
ffers" directive instead in /tmp/nginx/nginx-cfg2825306115:146
        nginx: [warn] the "http2_max_field_size" directive is obsolete, use the "large_client_header_buffers" directive inste
ad in /tmp/nginx/nginx-cfg2825306115:146
        2022/06/29 19:54:09 [warn] 179#179: the "http2_max_header_size" directive is obsolete, use the "large_client_header_b
uffers" directive instead in /tmp/nginx/nginx-cfg2825306115:147
        nginx: [warn] the "http2_max_header_size" directive is obsolete, use the "large_client_header_buffers" directive inst
ead in /tmp/nginx/nginx-cfg2825306115:147
        2022/06/29 19:54:09 [warn] 179#179: the "http2_max_requests" directive is obsolete, use the "keepalive_requests" dire
ctive instead in /tmp/nginx/nginx-cfg2825306115:148
        nginx: [warn] the "http2_max_requests" directive is obsolete, use the "keepalive_requests" directive instead in /tmp/
nginx/nginx-cfg2825306115:148
        2022/06/29 19:54:09 [emerg] 179#179: unexpected "}" in /tmp/nginx/nginx-cfg2825306115:449
        nginx: [emerg] unexpected "}" in /tmp/nginx/nginx-cfg2825306115:449
        nginx: configuration file /tmp/nginx/nginx-cfg2825306115 test failed
```

Symptoms: Ingress resources fail to route traffic to backend services or exhibit unexpected behavior due to misconfigured or invalid Ingress controller settings.

Diagnosis: Review Ingress controller logs (`kubectl logs <ingress_controller_pod>`) and Ingress resource definitions (`kubectl get ingress`) for any errors or inconsistencies.

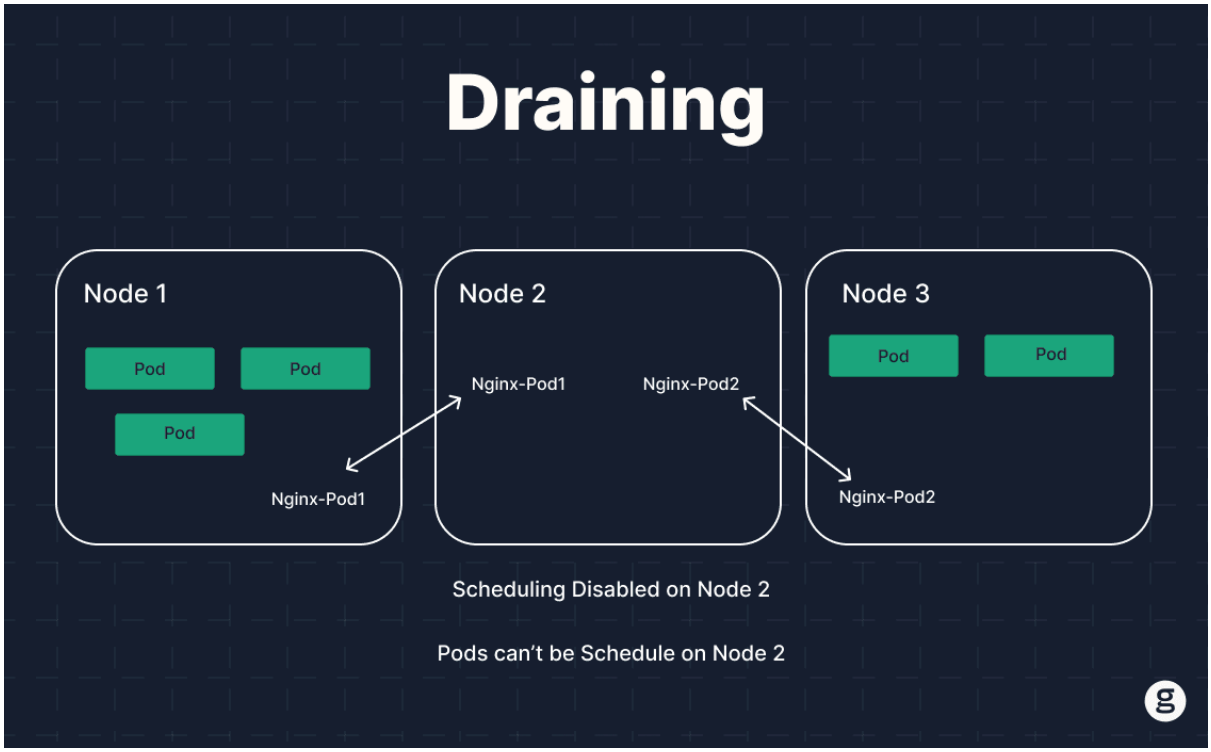
Solution:

1. Verify that the correct backend services are specified in the Ingress resource and that service endpoints are reachable from the cluster.
2. Check for typos or syntax errors in the Ingress resource annotations and ensure they conform to the Ingress controller's configuration requirements.
3. Monitor HTTP response codes and traffic patterns at the Ingress controller level to identify routing errors or misconfigurations.
4. Update Ingress controller configuration settings (e.g., TLS termination, path-based routing) based on application requirements and traffic management policies.

Scenario 28: Node Drain and Pod Eviction Handling

Symptoms: Nodes experience maintenance or upgrade events, triggering pod evictions and disrupting workload availability.

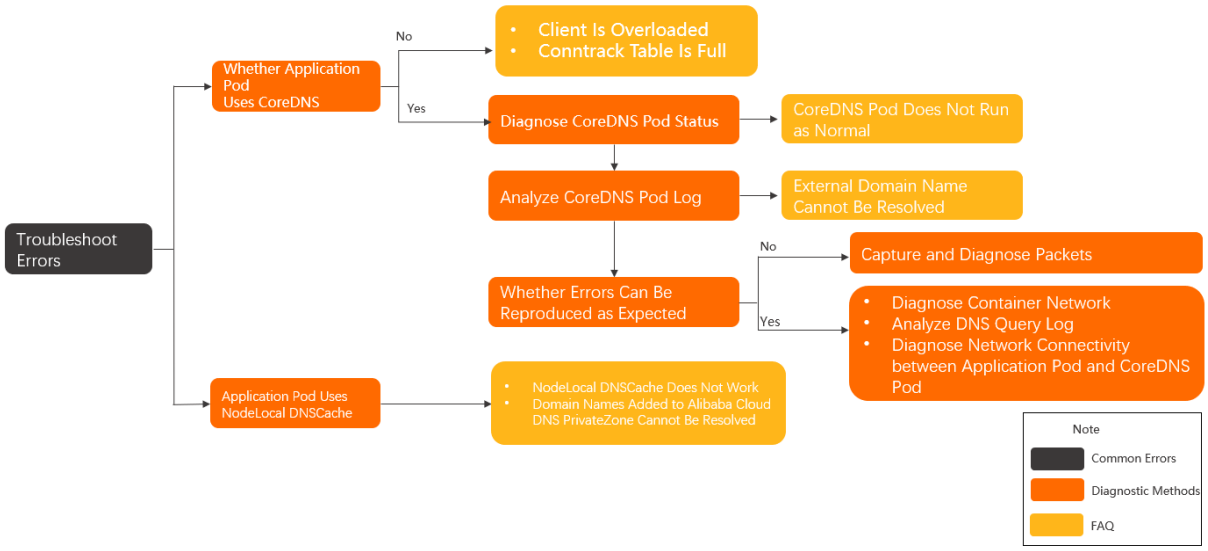
Diagnosis: Monitor Kubernetes node events (`kubectl get events --field-selector type=Warning`) and review node drain logs for any errors or warnings.



Solution:

1. Implement node drain procedures using tools like kubectl drain or node maintenance controllers to gracefully evict pods and reschedule them onto other nodes.
2. Configure PodDisruptionBudgets (PDBs) to limit the number of simultaneous pod disruptions during node maintenance activities and ensure high availability.
3. Verify that critical workloads have appropriate anti-affinity policies and node selectors to prevent all pods from being evicted from a single node.
4. Plan maintenance windows and communicate maintenance schedules to cluster users to minimize disruptions and downtime.

Scenario 29: Cluster-wide DNS Resolution Issues



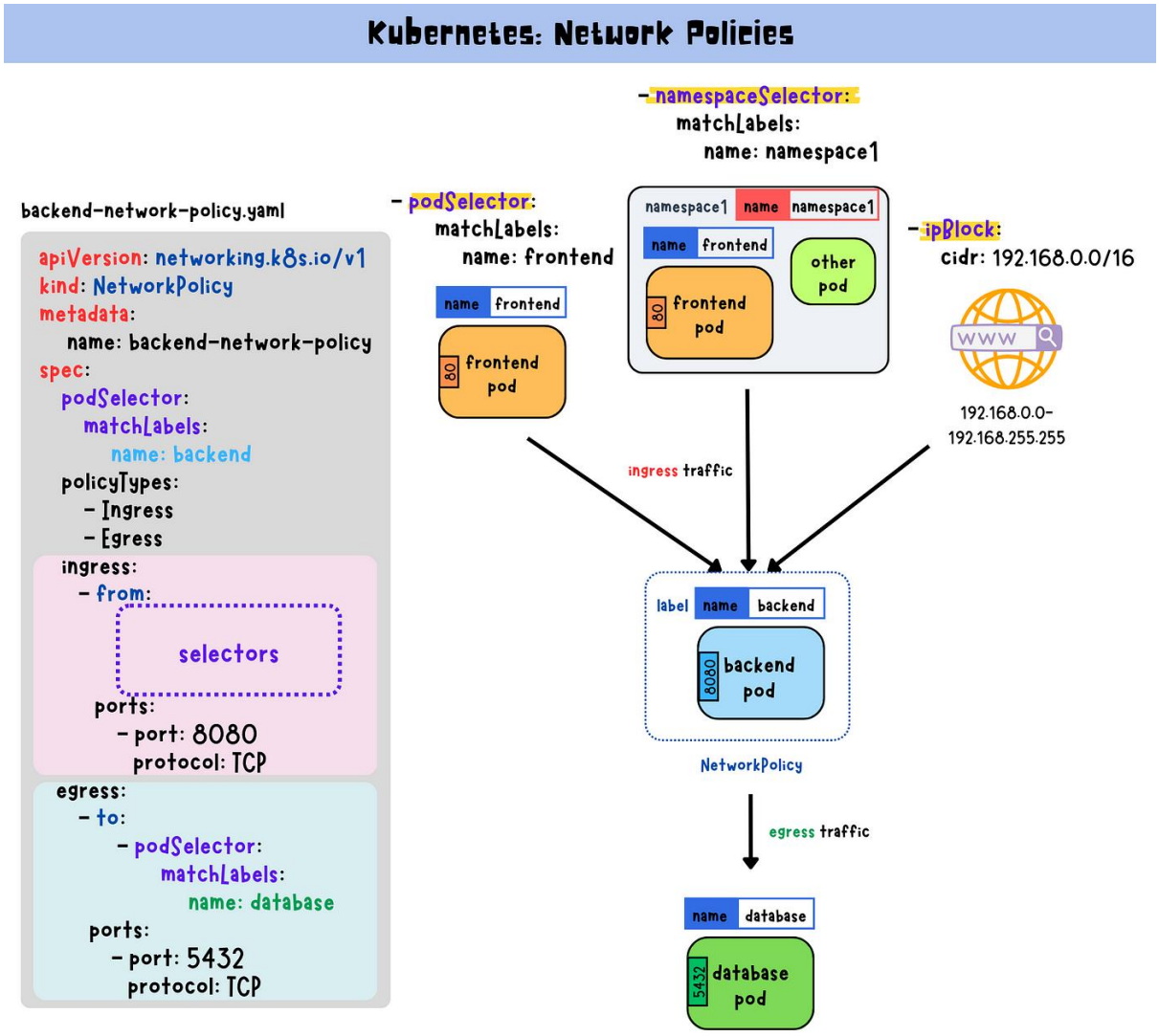
Symptoms: Pods fail to resolve DNS names or experience intermittent DNS resolution failures, impacting communication between services.

Diagnosis: Check CoreDNS or kube-dns logs (`kubectl logs -n kube-system <coredns_pod>` or `kubectl logs -n kube-system <kube-dns_pod>`) for DNS-related errors or warnings.

Solution:

1. Verify DNS configuration settings in CoreDNS or kube-dns ConfigMaps and ensure they include correct upstream DNS server addresses and search domains.
2. Monitor DNS query latency and packet loss metrics to identify network connectivity issues or DNS server performance degradation.
3. Implement DNS caching or local DNS resolution mechanisms to reduce reliance on external DNS servers and improve DNS resolution speed.
4. Use DNS debugging tools like dig or nslookup to troubleshoot DNS resolution problems and verify DNS records and responses.

Scenario 30: Network Policy Enforcement Failures



Symptoms: Network policies fail to enforce ingress or egress rules, allowing unauthorized network traffic between pods or namespaces.

Diagnosis: Review network policy resources (`kubectl get networkpolicy`) and network plugin logs (e.g., Calico, Cilium) for any errors or policy enforcement failures.

Solution:

1. Verify that the network plugin is correctly installed and configured to support network policies in the Kubernetes cluster.
2. Ensure that pods and namespaces are correctly labeled with appropriate selectors to match network policy rules.
3. Test network policy enforcement by applying traffic restrictions and monitoring network traffic using tools like tcpdump or Wireshark.
4. Review Kubernetes API server logs and audit logs to identify any security policy violations or unauthorized API requests that may bypass network policy enforcement.

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)