KubeCon | CloudNativeCon
Europe 2026

Banking on Reliability

Cloud Native SRE Practices in Financial Services

Clément Nussbaumer

clement.n8r.ch

Outline

1. SLOs as a Driver

Data-driven reliability improvements

2. Open-Source Monitoring

DNS monitoring & kubenurse mesh checks

3. End-to-End Testing

In-cluster Golang test suite

4. The 502 Mystery

Tracking down 6-per-million errors

About Me

👨‍💻 Clément Nussbaumer

- Systems Engineer at PostFinance 🏦 🇨🇭
- Living on a farm 🐄
- Musician in a Brass Band 🎺
- Father of 1 👶

🏦 PostFinance

- Systemic Swiss financial institution
- ~35 Kubernetes clusters
- Air-gapped environment
- Strict regulatory requirements

🖥️ The Platform

- 5+ years in production
- kubeadm / Debian clusters
- ongoing Talos + TOPF migration

Every failed request = potential denied payment 💸

Part 1: Service Level Objectives (SLOs) as a Driver

From "it feels slow" to data-driven reliability

- "The cluster feels slow today" 🤷 — devs complaining, sluggish **k9s** sessions
- Basic Grafana dashboards, but no clear target
- Degraded performance for **months** — never properly investigated

> Without a target and a timeline, "slow" is subjective and easy to ignore.

API Server SLOs

Three SLOs defined for the Kubernetes API server[1]:

- **Availability** — less than 0.1% of requests return 5xx or 429
- **Latency (read)** — GET/LIST within threshold (varies by subresource & scope)
- **Latency (write)** — POST/PUT/PATCH/DELETE within **1s**

Defining the PromQL queries was tedious but **sloth**[2] made it tractable:

```
slos:
  - name: apiserver-availability
    objective: 99.9
    sli:
      events:
        error_query: sum(apiserver_request_total{code=~"5..|429"})
        total_query: sum(apiserver_request_total)
```

→ generates all recording rules, multi-window burn-rate alerts, error budget calculations

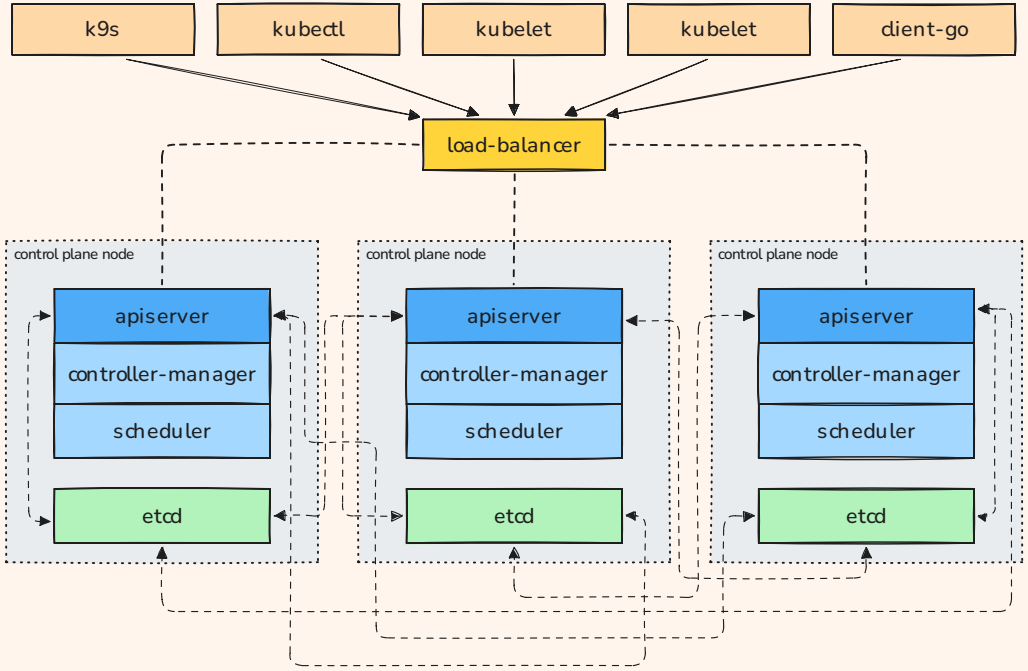
1. https://sre.google/sre-book/service-level-objectives/
2. https://github.com/slok/sloth

SLOs Reveal the Truth

> "the cluster feels slow" → "we burned 40% of our error budget during Tuesday's upgrade"

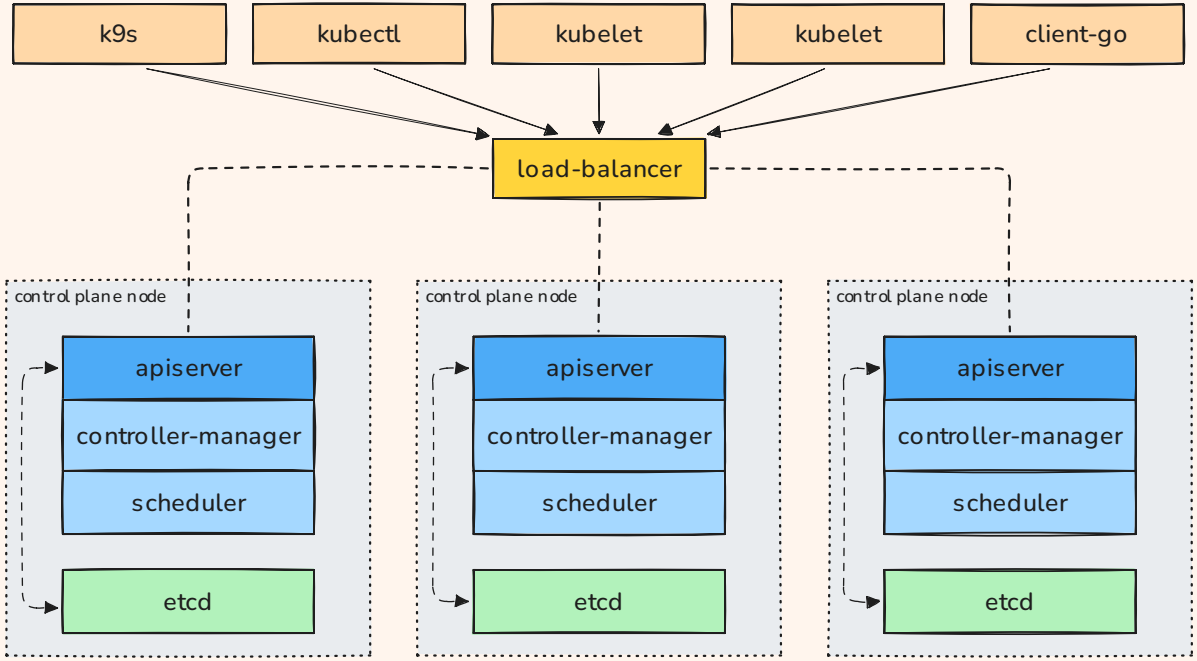
Fix #1: etcd Topology — Before

Each apiserver connects to all 3 etcd members



Fix #1: etcd Topology — After

Stacked topology: each apiserver talks to its local etcd[1]



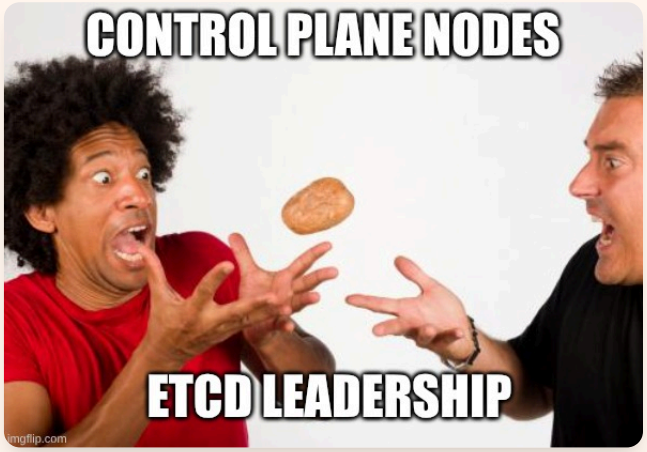
1. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/

Fix #2: etcd Leadership Migration

Moving leader away before maintenance

- Before upgrading a node, move etcd leadership to another member
- Avoids leader election during maintenance window
- Light improvement, but not the full solution

```
# move etcd leadership away from the node being upgraded
etcdctl move-leader $NEW_LEADER_ID
```



Fix #3: The Real Culprit

One CP node doing all the work, the other two idle.

- **Long-lived HTTP/2 connections** never redistributed
- Clients open a connection once → reuse it forever

The fix: `--goaway-chance=0.001` [1]

→ 1/1000 requests get a GOAWAY, client reconnects through LB

RFC 7540 §6.8 — GOAWAY[2]

The GOAWAY frame (type=0x7) is used to initiate shutdown of a connection.

GOAWAY allows an endpoint to **gracefully stop accepting new streams** while still **finishing processing of previously established streams**.

1. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/
2. https://datatracker.ietf.org/doc/html/rfc7540#section-6.8

Part 2: Open-Source Monitoring Tools

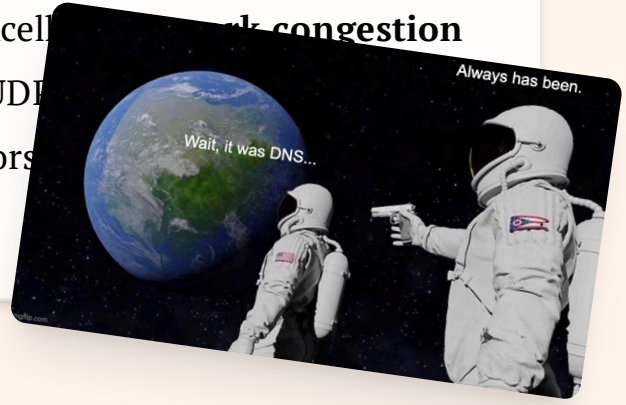
Built from production needs, improved by the community

🏥 kubenurse[1]

- DaemonSet performing continuous network health checks
- Node-to-node, apiserver, ingress, service & DNS connectivity
- Latency histograms + error counters with httptrace events

🔍 hostlookuper[2]

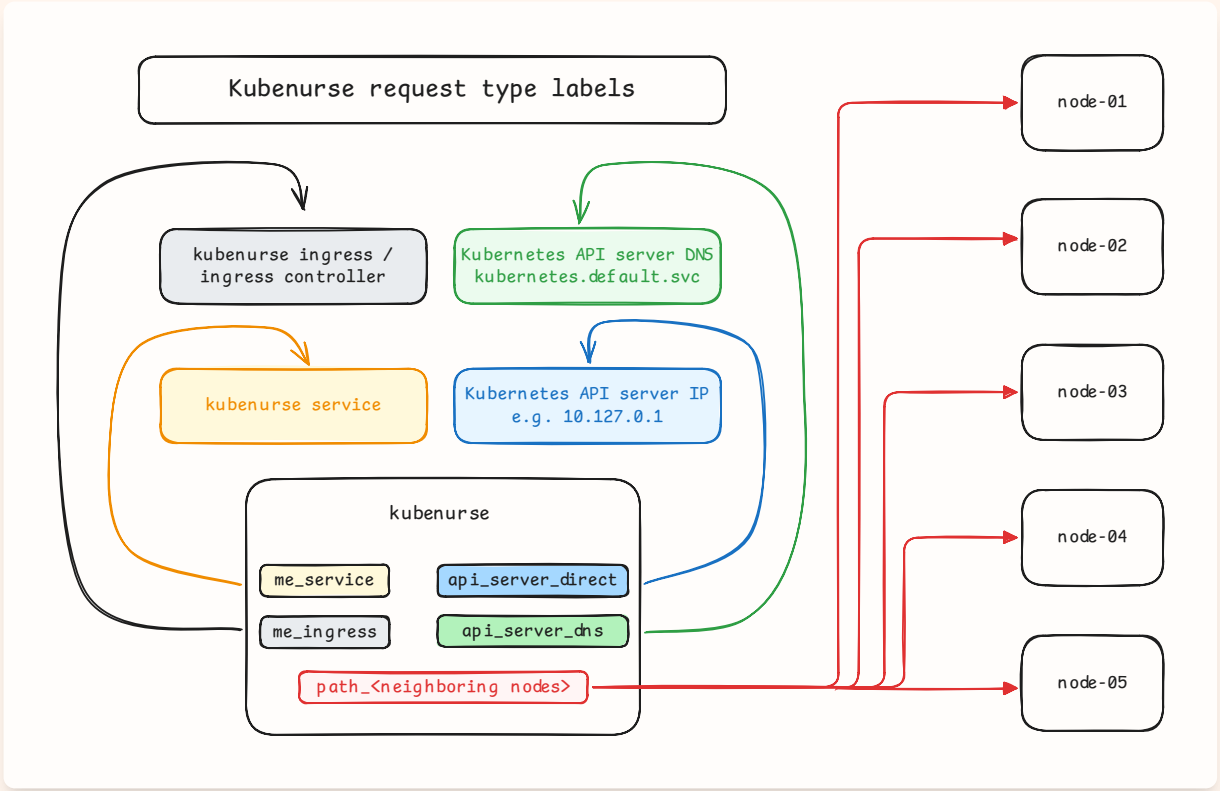
- Periodically resolves DNS targets, exports latency + errors as Prometheus metrics
- DNS is an excell… …**k congestion indicator**: UDP… result in errors…



1. https://github.com/postfinance/kubenurse
2. https://github.com/postfinance/hostlookuper

kubenurse: 5 Request Types

A single DaemonSet, n network paths checked every x seconds

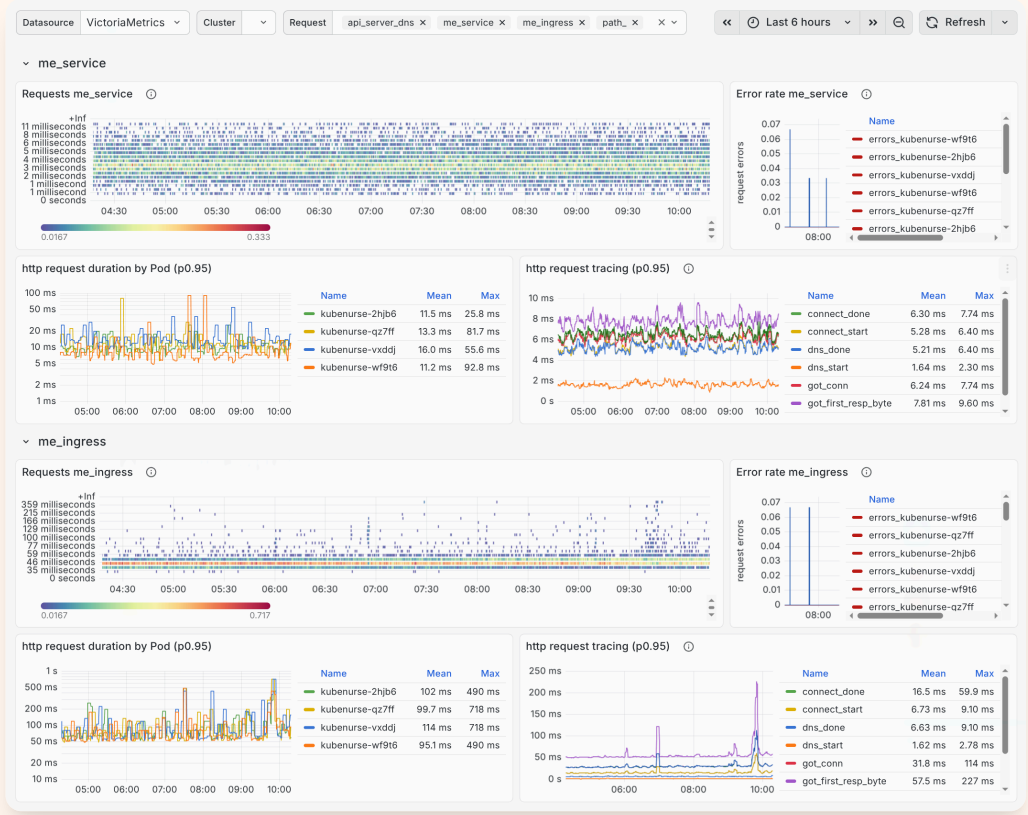


kubenurse: httptrace Instrumentation

Metrics are labeled with **httptrace event types** — precise breakdown of each request phase:

- `dns_start` / `dns_done`
- `connect_start` / `connect_done`
- `tls_handshake_start` / `tls_handshake_done`
- `got_conn` / `got_first_response_byte`

> On errors, the **event label** tells you exactly which phase failed



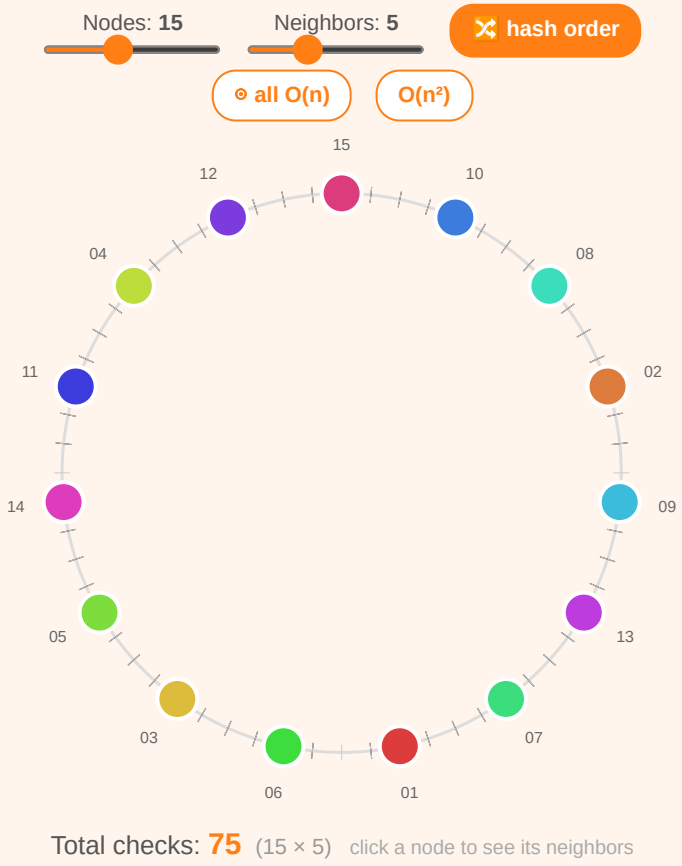
kubenurse: $O(n^2) \rightarrow O(n)$

Community discussion via GitHub issue #55 sparked the fix

The problem: every pod checked every other pod → $O(n^2)$

The fix: deterministic neighbor filtering

- Node names are hashed → each pod checks n **neighbors** based on hash order (default: $n = 10$)
- Distribution is random but deterministic → stable metrics across restarts



Graceful Shutdown: a Generic Problem

SLOs on kubenurse revealed errors on the `me_ingress` check during upgrades/maintenance

The problem: not specific to ingress-nginx — affects **any** process behind a load balancer

`SIGTERM` arrives, but the LB doesn't know yet → requests still routed to a dying process → errors

The fix: lameduck shutdown[1] (inspired by CoreDNS)

- On `SIGTERM` : **keep serving** for a few seconds (default: **5s**)
- LB / proxies / CNI (hopefully) catch up and stop sending traffic
- Then stop the server

1. https://github.com/postfinance/kubenurse/commit/cef5f2ef

Part 3: Continuous End-to-End Testing

Your end users should NOT be your end-to-end tests

Why?

- Complex interactions between components
- Networking, storage, security, DNS validations

Our approach

- Go test suite using `e2e-framework` [1],
  scheduled as a **K8s CronJob** (every 15min)
- Results captured with **OpenTelemetry** →
  Grafana dashboards

```go
func TestKubernetesDeployment(t *testing.T)
    start := time.Now()

    t.Cleanup(func() {
      metricsCollector.RecordTestExecution(
        t, time.Since(start),
      )
    })

    dep := newDeployment("nginx", 3)
    err := env.Create(ctx, dep)
    require.NoError(t, err)

    waitForPodsReady(t, dep, 30*time.Second)
}
```

1. https://github.com/kubernetes-sigs/e2e-framework

Open-Source: e2e-tests[1]

An analogous open-source implementation you can try today

Test	What it validates
Deployment	Pod scheduling, container runtime, workload lifecycle
Storage (CSI)	PV provisioning, read/write operations
Networking	DNS resolution, service discovery, inter-pod connectivity
RBAC	Role-based access boundaries, permission enforcement

How it runs

- Deployed as a K8s **CronJob** (every 15min)
- Runs in-cluster: creates a namespace, provisions test resources, validates everything works
- Metrics stream to OTLP endpoint → VictoriaMetrics / Grafana

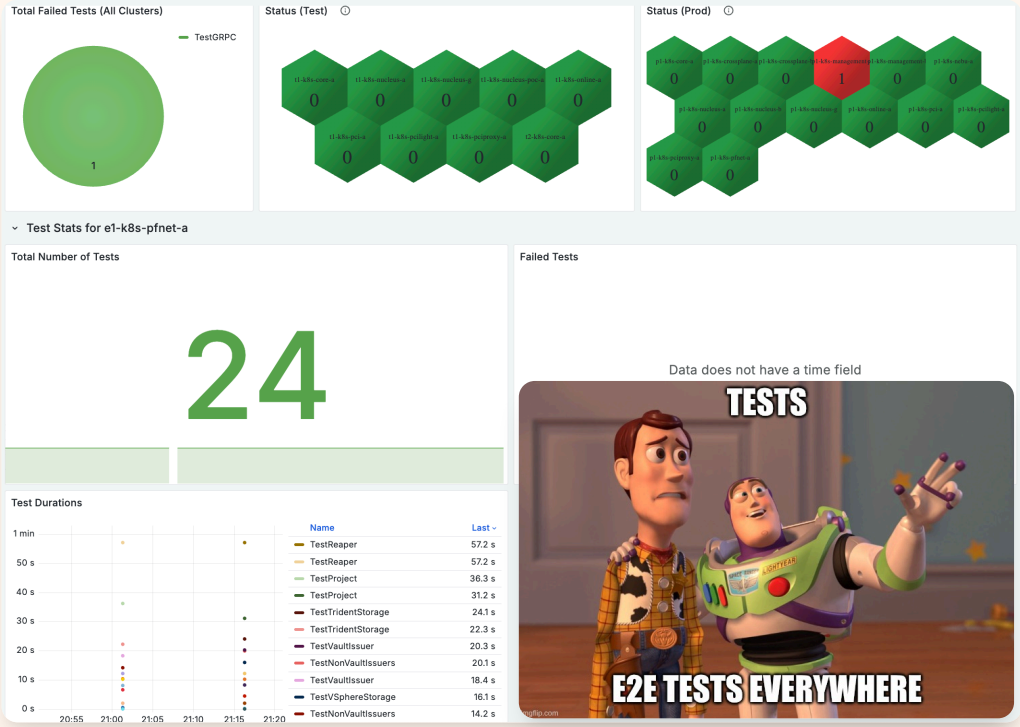
Fork it, adapt the tests to your platform, deploy as a CronJob → instant cluster health monitoring

1. https://github.com/clementnuss/e2e-tests

E2E Test Results

- Tests run every **15 minutes** across all clusters
- Results visualized per cluster, per test category

> Alert rules trigger on test failures → we know before users do



Part 4: The 502 Mystery

Tomcat-based e-finance app — 502s caused real user impact when retrieving e-banking resources

~1.7M requests/day on one ingress. **8–10 failures per day.** Where would you look?

- 502s uniformly distributed across all ingress-nginx pods
- No pattern in time, endpoint, or client
- App pods healthy, no errors in application logs
- CPU / memory fine, network policies correct
- Load testing with **K6** couldn't reproduce it
- Errors correlate with request volume — more traffic, more 502s
- But the **rate** stays constant: ~6 per million

claper.n8r.ch · code: **#KUBECON**

The Investigation

The breakthrough: finding the right log line

ingress-nginx error logs contain the **FQDN**, not the ingress name — we were searching for the wrong thing.

```
upstream prematurely closed connection while reading response header from upstream
```

This tells us:

- nginx **had** an open connection to the upstream (backend pod)
- It sent a request on that connection
- The backend **closed the connection** before responding
- nginx has no response to return → **502 Bad Gateway**

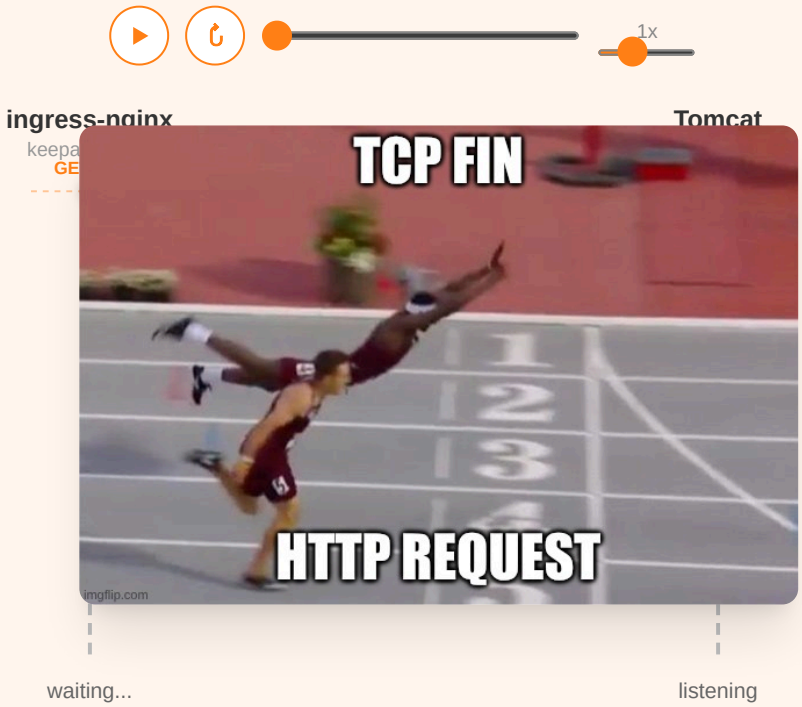
The Race Condition

Two conflicting keepalive timeouts:

- **nginx**: keeps connections open for **60s** (default)
- **Tomcat**: closes idle connections after **20s** (default)

The race window:

1. Connection sits idle for ~20s
2. Tomcat sends **FIN** to close it
3. At the same moment, nginx sends a **new request** on that connection
4. → **502 Bad Gateway**



The Fix[1]

One environment variable:

```
export TC_HTTP_KEEPALIVETIMEOUT="75000"  # 75s > nginx's 60s
```

The rule: upstream `keepalive_timeout` must be **greater** than the reverse proxy's

- nginx default: **60s**
- Tomcat was: **20s** → now **75s**
- Backend always outlives the proxy's connection → no more race

Lessons

- K6 load tests failed because they didn't test **idle + burst** patterns
- Connection lifecycle timeouts must be coordinated **end-to-end**
- The fix is trivial — finding it is the hard part

1. https://clement.n8r.ch/en/articles/502-upstream-errors/

Key Takeaways

SLOs are a forcing function

From "it feels slow" to data-driven fixes

Open-source your tools

The best fixes and discussions come from the community — not always code, sometimes just the right conversation

Test continuously, in-cluster

Your end users should not be your e2e tests

Every error matters

8 out of 1.7M — still worth fixing

Thank you!

kubenurse

Network monitoring DaemonSet

e2e-tests

K8s cluster validation CronJob

502 blog post

Full debugging walkthrough

clement.n8r.ch

Questions? Ask here!

claper.n8r.ch · code: **#KUBECON**

Appendix: Reproducing the 502 with K6

The key: **idle** → **burst** stages with varying idle durations to hit the keepalive race window

```
// Cycle: ramp up → sustain → ramp down → idle
// Idle duration increases (4s→11s) to maximize
// chance of hitting Tomcat's 20s timeout boundary
function generate_stages() {
    var stages = []
    for (let i = 4; i < 12; i++) {
        stages.push({ duration: "5s", target: 100 });
        stages.push({ duration: "55s", target: 100 });
        stages.push({ duration: "5s", target: 0 });
        stages.push({ duration: i + "s", target: 0 });
    }
    return stages
}
```

```
export let options = {
    noConnectionReuse: true,
    noVUConnectionReuse: true,
    scenarios: {
        http_502: {
            stages: generate_stages(),
            executor: 'ramping-vus',
            gracefulRampDown: '1s',
        },
    },
};

export default function() {
    let data = { data: 'Hello World' };
    for (let i = 0; i < 10; i++) {
        let res = http.post(
          `${__ENV.URL}`, JSON.stringify(data));
        check(res, {
          "status was 200": (r) => r.status === 200
        });
    }
    sleep(1);
}
```