Kong / K8s Lab — Lessons from 2026-09-13 to 09-15

A running log of root causes and fixes from this lab, organized so future-you can self-diagnose without re-explaining the whole cluster from scratch.

---

1. Pulumi `ConfigFile` deletes anything not in the current file — this bit us twice

Symptom: After switching a config value (e.g. `routeManifest` or `observabilityManifest`) to a different YAML file and running `pulumi up`, an *unrelated* namespace and everything in it silently disappeared.

Root cause: `kubernetes:yaml/v2:ConfigFile` tracks every object inside the file as a child resource. On each `pulumi up`, Pulumi diffs the *previous* file's object list against the *current* file's object list. Anything missing from the new file is **deleted** — even if it was never meant to be touched. This is standard declarative IaC behavior, not a bug.

Fix: Each independently-lifecycled group of resources needs its **own** `ConfigFile` component/config key. In `Pulumi.yaml` we have two: `routeManifest` (drives `kongLesson`) and `observabilityManifest` (drives `observability`). **Always double check which config key you're setting before `pulumi up`:**

```bash
pulumi config get routeManifest
pulumi config get observabilityManifest
```

Route-layer lesson files (echo, rate-limit, auth, IP-restriction) → `routeManifest`. Cross-cutting infra (Prometheus, and future Jaeger/Loki/Vault/etc.) → `observabilityManifest` (or a new dedicated key per tool, if they shouldn't share a lifecycle either).

If a resource needs to move from one component to the other (like Prometheus did once, by accident), cluster-scoped objects (`Namespace`, `ClusterRole`, `ClusterRoleBinding`, `PersistentVolume`) will hit an "already exists" error on the create side before the old owner's delete has run. Fix with:

```bash
pulumi config set kubernetes:upsertExistingObjects true
pulumi up
```

And if state and reality diverge after a partial failure (e.g. a namespace got deleted mid-migration, taking resources from *both* components with it):

```bash
pulumi refresh --yes
pulumi up
```

Prevention: Before any `pulumi config set` + `pulumi up`, ask: *which component does this file belong to, and does the file still contain everything that component currently owns?* Treat every file swap as a full-file diff, not an incremental change.

---

2. Kong (DB-less/InMemory) does not share config with Kubernetes — always verify at both layers

Symptom: "no Route matched with those values" persisting even after the Ingress/KongPlugin objects were confirmed deleted from Kubernetes.

Root cause: Kong Gateway here runs `update_strategy: InMemory` — config lives only in each gateway pod's process memory. The Ingress Controller has to push a separate sync to **every** gateway replica's Admin API individually. A sync gap (leader-election hiccup, a replica that just started) means Kong's live state can lag behind Kubernetes' declared state in either direction.

Fix — check Kong's actual runtime state, not just the K8s objects:

```bash
kubectl -n kong port-forward pod/<gateway-pod-name> 8444:8444
curl -sk https://localhost:8444/routes    | jq '.data[] | {id, paths}'
curl -sk https://localhost:8444/plugins   | jq '.data[] | {id, name, route, service}'
```

If it disagrees with what Kubernetes says, force a resync:

```bash
kubectl -n kong rollout restart deployment/kong-d19f3a41-controller
```

Prevention: Whenever "it should be gone/there but isn't behaving that way," check **both** layers — `kubectl get <resource>` AND the Admin API of *each* gateway pod — before assuming the manifest is wrong.

---

3. Scaling Kong's gateway to 2 replicas didn't give redundancy — check node spread, not just replica count

Symptom: After `kubectl scale deployment kong-d19f3a41-gateway --replicas=2`, still got occasional 404s.

Root cause: Both replicas landed on the same node (`k8worker1`) — no pod anti-affinity was set, so the scheduler didn't spread them. Real load-balancing/redundancy step (#8 on the learning list) wasn't actually achieved yet.

Follow-up TODO: Add a `podAntiAffinity` (or `topologySpreadConstraints`) to the gateway Deployment before trusting multi-replica behavior for real testing.

---

4. `externalTrafficPolicy: Local` + single-replica Service = node-specific reachability

Symptom: `curl https://<LB-VIP>/echo` worked from `k8master` but got instant connection-refused from the host machine.

Root cause: Cilium L2 Announcement was answering ARP for the VIP from whichever node currently holds the lease. With `externalTrafficPolicy: Local`, a node **without a local backend pod** for that Service will refuse *external* traffic outright — even though internal (node-to-node) traffic still gets load-balanced normally regardless of the policy. Kong's gateway pod only existed on `k8worker1`; `k8master` had nothing local to serve it.

Fix (pick one):

- `kubectl patch svc <svc> -n kong -p '{"spec":{"externalTrafficPolicy":"Cluster"}}'` — simplest, but loses real source-IP preservation.
- Scale/spread the gateway so every node has a local pod (see #3 above) — the "correct" fix, preserves source IP.

Prevention: Any time "works from inside the cluster, fails from outside," suspect `externalTrafficPolicy: Local` + uneven pod placement before suspecting DNS/firewall/routing.

---

5. LXD `local` PersistentVolume paths must match the *container's* view, not the host's

Symptom: Pod stuck in `ContainerCreating`/mount failure; `ls <path>` from inside the LXD container returned "No such file or directory" even though `findmnt`/`zfs list` on the host showed the path mounted fine.

Root cause: `k8master` is an LXD **container**, not the bare host. A ZFS dataset mounted on the host filesystem is invisible inside the container unless explicitly passed through via an LXD disk device — and that device's `path:` field can differ from its `source:` field (ours mapped `/srv/prometheus-data` on the host to `/mnt/prometheus-data` inside the container).

Diagnostic sequence:

```bash
lxc config device show k8master       # check existing device path vs source
lxc exec k8master -- ls -la <path>    # confirm visibility from inside the container
```

Prevention: For any `local` PV on this lab, the `spec.local.path` in the PV YAML must match the **container-side** path from `lxc config device show`, not the host-side `source:`. Check this *before* writing the PV manifest, not after a failed apply.

---

6. Control-plane node taint blocks ordinary pods — even with a matching `nodeSelector`

Symptom: Pod stuck `Pending`, `FailedScheduling: node(s) had untolerated taint(s)`, despite `nodeSelector` correctly targeting `k8master`.

Root cause: `nodeSelector`/`nodeAffinity` only says *where a pod is allowed to go*; it says nothing about *taints*, which are a separate opt-in mechanism. Control-plane nodes carry `node-role.kubernetes.io/control-plane:NoSchedule` by default — Kong's own pods tolerate this (baked into its Helm chart), which is why they scheduled fine on the same node while a hand-written Deployment didn't.

Fix — add explicitly to any Deployment pinned to `k8master`:

```yaml
tolerations:
  - key: node-role.kubernetes.io/control-plane
    operator: Exists
    effect: NoSchedule
```

Prevention: Any time a Deployment is deliberately pinned to `k8master` via `nodeSelector`, add this toleration up front — don't wait for the scheduling failure.

---

7. ZFS pool `SUSPENDED` ≠ disk-full — and `zpool clear` cannot fix a device the kernel has disowned

Symptom: SSH to `k8master` hung indefinitely after auth succeeded (`Entering interactive session` then nothing) — even `ssh k8master 'echo hello'` hung.

Root cause chain:

1. `masterpool`'s backing disk (`sdc`, a `VBOX HARDDISK` virtual disk) started throwing I/O errors, then the kernel issued `ata4: disable device` and returned `hostbyte=DID_BAD_TARGET` on every subsequent I/O — i.e. the kernel **removed the device from the SCSI subsystem entirely**, not just "erroring."
2. ZFS responded by `SUSPEND`ing the pool to prevent corruption — freezing *all* I/O against it.
3. `k8master`'s entire rootfs lives on `masterpool`, so every process (including `sshd`, which needs to write `wtmp`/`lastlog` on login) blocked forever.
4. `zpool clear masterpool` failed with a plain `I/O error` — clearing errors on a device that no longer exists at the kernel level is a no-op; the device has to physically reappear first.
5. Graceful `lxc stop` also hung, because processes blocked on dead-disk I/O enter uninterruptible sleep (`D` state) — **unkillable by any signal, including SIGKILL**, until the I/O resolves or the box reboots.

Diagnostic sequence (fastest path to the real cause, in order):

bash

```
ssh -vvv <host>                                  # find exactly where it hangs
ssh <host> 'echo hello'                          # rules shell-startup scripts in/out
lxc exec <container> -- df -h /                  # rules disk-full in/out
zpool status                                      # SUSPENDED/FAULTED jumps out immediat
sudo dmesg -T | grep -iE '<disk>|ata|i/o error'  # kernel-level root cause (DID_BAD_TARG
```

Recovery sequence:

bash

```
lxc stop <container> --force        # graceful stop will also hang — don't wait on it
sudo reboot                         # only real fix once kernel has disowned the device
# --- after reboot ---
lsblk | grep <disk>                 # confirm device is back
zpool status <pool>                 # confirm ONLINE, 0 0 0 errors
sudo zpool scrub <pool>             # verify actual data integrity, not just "device resp
zpool status <pool>                 # wait for "scrub repaired 0B ... with 0 errors"
lxc start <container>
```

Prevention / mental model for next time: SSH hanging *after* successful auth is almost never an SSH problem — it's the remote shell/session blocked on something else (disk I/O, PAM/dbus, resource exhaustion). Check `zpool status` and `df -h` on the target before touching SSH config or trying to restart sshd.

Note on this specific incident: `sdc` is a VirtualBox virtual disk (`VBOX HARDDISK`), and it came back `ONLINE` with 0 errors and a clean scrub immediately after reboot — consistent with a transient VirtualBox storage-layer hiccup on the host side, not a genuinely failing physical drive. If this recurs, check host-side disk space and the VM's VirtualBox event log before assuming hardware failure.

---

8. General debugging order that would've saved time today

1. **Check the actual error/state first** (`kubectl describe`, `zpool status`, `dmesg`) before assuming the layer you were just working in is the cause.
2. **Separate "network reachable" from "app configured" from "data intact."** A 404 and a connection-refused and a hang are three different failure classes — don't reach for the same fix for all three.
3. **After any multi-day-old Ingress/resource shows unexpectedly missing, check `AGE`** — if something's been gone for 39 hours, today's reboot isn't the cause.
4. **A clean reboot does NOT wipe Kubernetes/etcd state** — if something's missing after a reboot, it was almost certainly already broken beforehand; the reboot just surfaced it.
5. **Kong's route config is separate from Kubernetes' Ingress objects** (DB-less/InMemory mode) — check both when something doesn't seem to sync.