KubeCon CloudNativeCon North America 2026

Autoscaling Reality Check: The Hands-On HTTP Optimization Challenge

Salt Lake City, Utah, United States



David Pech

Who am I



David Pech

wrike

GOLDEN
Kubestronaut

Once upon a time…

there was a demand to run the cluster *more efficiently*. So you ask around:

you: how do I autoscale my app on Kubernetes?

YouTube / ChatGPT / your vendor: super easy!

```
kubectl autoscale deployment app --cpu-percent=80
```

nodes, Pods — it all scales seamlessly 🎉

> The demo app: nginx, ~5 MB of RAM, ready in milliseconds, one endpoint, one kind of request.

…but maybe it's not such a fairy tale

Your app

- takes a minute or two to start
- *Ready* ≠ ready — caches still warming up
- sidecars with their own startup and resources
- fast *and* slow requests on the same endpoint

Your cluster

- a new node takes **2–10 minutes**, not 30 s
- scaling up is easy → **90 % of capacity unused**
- Black Friday: peak = 10–20× baseline
- a new release changes the performance profile — one weekend ate months of savings

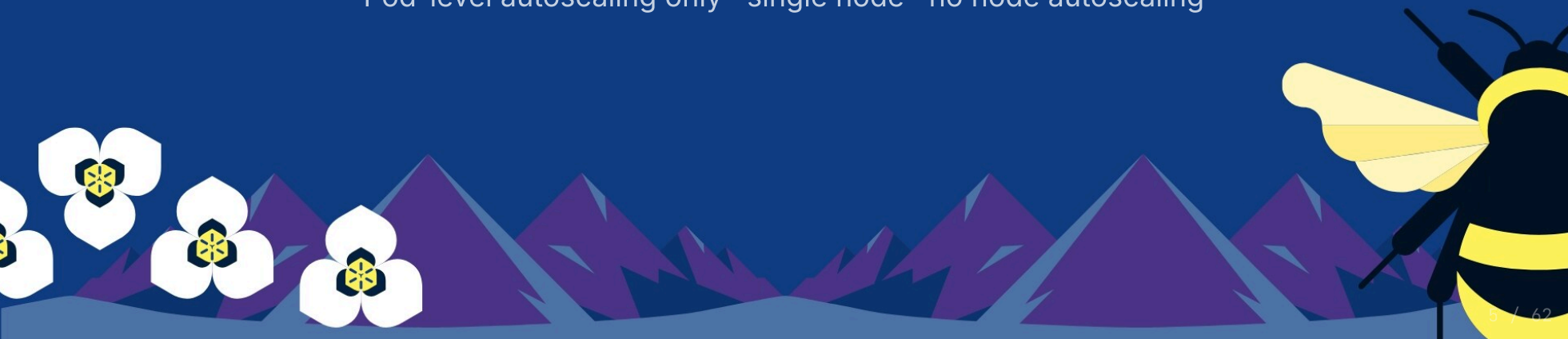
> Scale up and scale down are **different disciplines**. And every new tool (KEDA, Knative, a service mesh) is something you will run for years.

KubeCon CloudNativeCon North America 2026

Let’s get some hands-on

How fast can your Pods *really* scale when the traffic hits?

Pod-level autoscaling only · single node · no node autoscaling



Get a cluster

Local k3d — Docker with ~4 CPUs / 8 GB free

```
git clone https://github.com/depeche-io/kubecon-2026-autoscaling-challenge
cd kubecon-2026-autoscaling-challenge
./bin/up.sh          # 5–10 minutes
```

GitHub Codespaces — *Code* → *Codespaces* → *Create codespace on main* (`up.sh` runs on start)

Killercoda — *Autoscaling Reality Check* scenario (tight for the full stack)

Argo CD `:30080` · Grafana `:30030` · Prometheus `:30090` — `admin` / `@utoSc4ling`

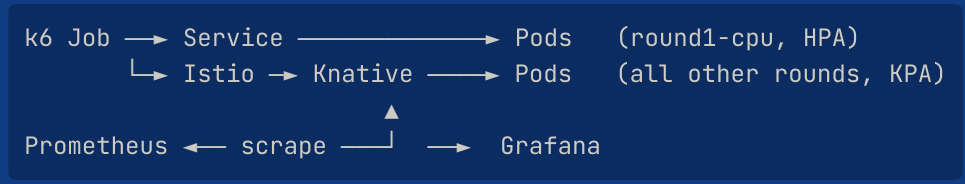
github.com/depeche-io/
kubecon-2026-autoscaling-challenge

The technology doesn't matter

What's in the box — all of it is **plumbing**:

- **Argo CD** — GitOps
- **Istio + Knative Serving** — the second autoscaler
- **kube-prometheus-stack** — 5 s scrapes, one dashboard
- **k6** — the load, as suspended CronJobs

Flux instead of Argo CD, KEDA instead of Knative, Datadog instead of Prometheus — **same lessons**.



What we care about: **which signal** you scale on, **how long** a new Pod takes to be useful, **what** is in a Pod, and **how requests wait**.

Six rounds, side by side

Every round is its own namespace + Argo CD app (`challenge-<round>`), all running at once.

Round	App	Autoscaler	Lesson
`round1-cpu`	nginx, static page	HPA, CPU 80 %	does CPU tell you anything?
`round2`	Go, 10 workers	Knative	saturation at ~0 % CPU
`round3`	nginx + php-fpm, one Pod	Knative	the Pod is the scaling unit
`round4`	Spring MVC, 30 s warm-up	Knative	startup time is your budget
`round5`	nginx, scale to zero	Knative	cold starts
`round6`	BFF → backend	Knative × 2	timeouts along the chain

Testing methodology — the benchmark

All scenarios call `http://app.<namespace>/?minWait=<min>&maxWait=<max>` (k6 `constant-vus`):

Scenario	Duration	Parallel (VUs)	Wait per request
`fast`	120 s	50	10 – 100 ms
`slow`	120 s	5	1 – 3 s
`all`	one after another, 240 s	50 / 5	mixed

Closed model: each VU sends the next request only after the answer → at most 50 requests in flight. A slow app gets *fewer* requests, not more.

5 s client timeout — anything slower counts as a failure.

Testing methodology — what we measure

User view (k6)

- availability = success / total
- p50 / p90 / p99 latency

Cost view (cluster)

- CPU efficiency = usage / limits
- memory efficiency = usage / limits
- avg. number of app Pods

Grafana → *Autoscaling Reality Check* → `namespace` = your round. Plus per-app panels: busy workers, queues, Knative desired vs. actual Pods.

Good autoscaling = **users don't notice** and you **don't pay for idle**.

Testing methodology — the loop

1. **Pick** the round — open its Argo CD app
2. **Run** — CronJob `bench-all` → ⋯ → **Create Job**
3. **Read** — Grafana, `namespace` = the round
4. **Tune** — autoscaling resource → *Live manifest* → *Edit*

   self-heal is off; *Sync* resets the round
5. **Rerun**

Leaderboard 🏆

Your runs over here

Round 1 · CPU HorizontalPodAutoscaler — setup

- `nginx:1.29-alpine` static page — *the waits are ignored*
- nginx: requests `25m`, limits `100m`
- plain Deployment + Service, no Knative

```yaml
# rounds/round1-cpu/hpa.yaml
minReplicas: 1
maxReplicas: 10
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 80
behavior:
  scaleDown:
    stabilizationWindowSeconds: 120
```

Round 1 · CPU HPA — expectations

- Does it scale at all? How many Pods at the peak?
- How long until the first new Pod serves traffic?
- 80 % of *what* exactly?
- Does p99 get better with more Pods?

KubeCon CloudNativeCon North America 2026

🎬 Demo time

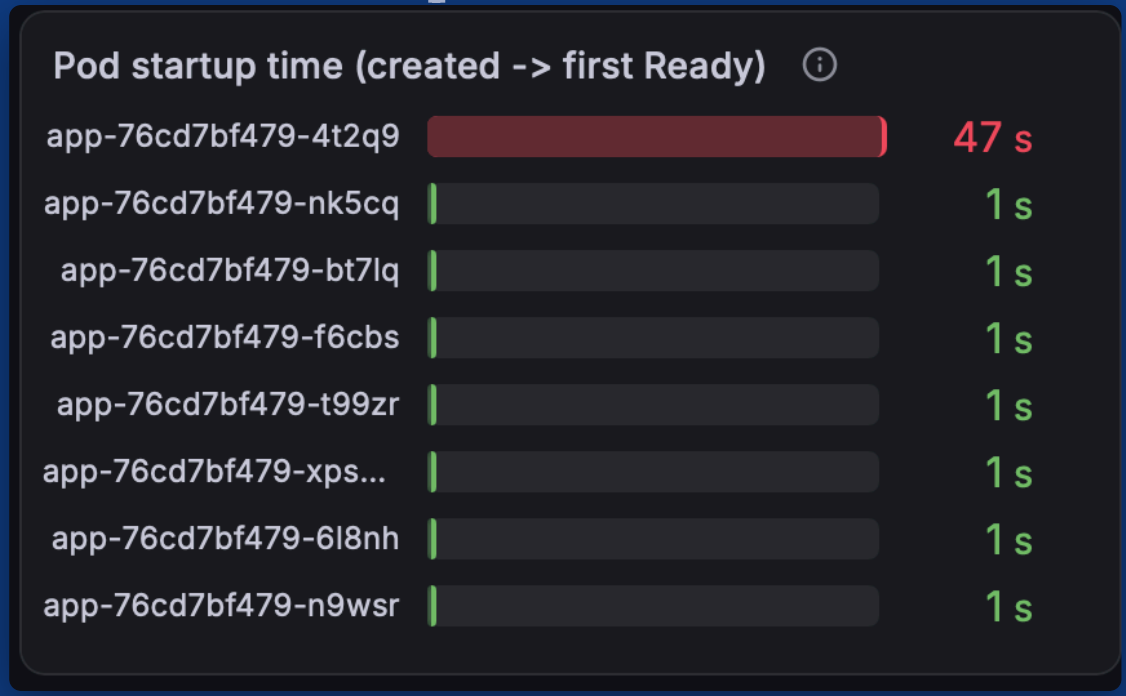
Round 1 · CPU HPA

Argo CD `challenge-round1-cpu` → `bench-all` → **Create Job** · Grafana `namespace=round1-cpu`



Round 1 · CPU HPA — results

Why is there 1 slow startup?



Round 1 · CPU HPA — results

Did all the Pods serve roughly the same number of requests?



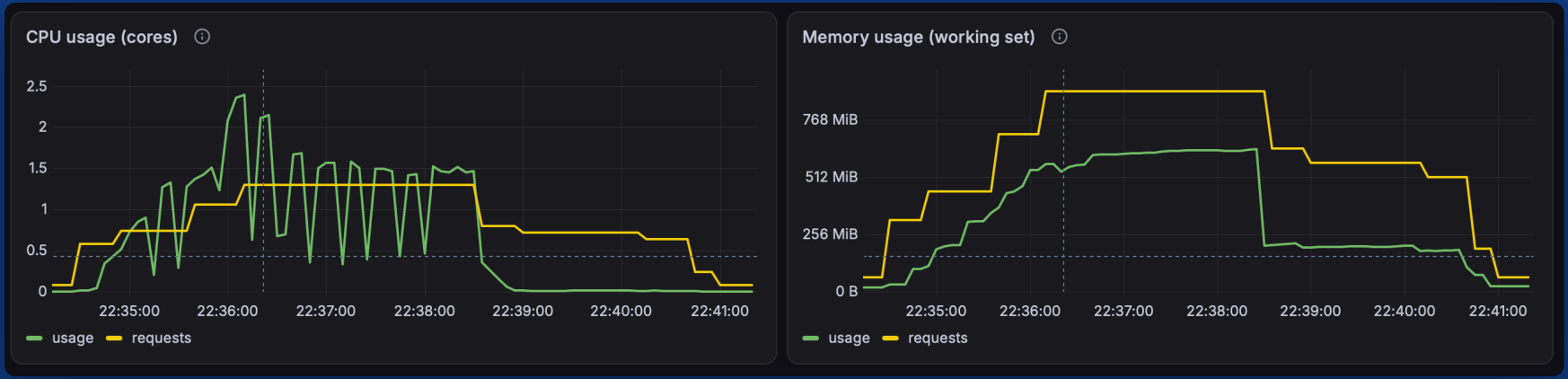
Round 1 · CPU HPA — results

Millions of requests served without any error.



Round 1 · CPU HPA — results

Why do we keep Pods running after the load test?



Round 1 · CPU HPA — how to improve

- **Right-size requests** — utilization is relative to them; wrong requests = wrong scaling
- drop the CPU **limit** — throttling adds latency, not throughput
- `behavior.scaleUp` policies — how many Pods per period
- higher `minReplicas` instead of reacting
- do you need to scale at all? One nginx with a full CPU beats ten throttled ones

Round 1 · Improved — before vs. after

After



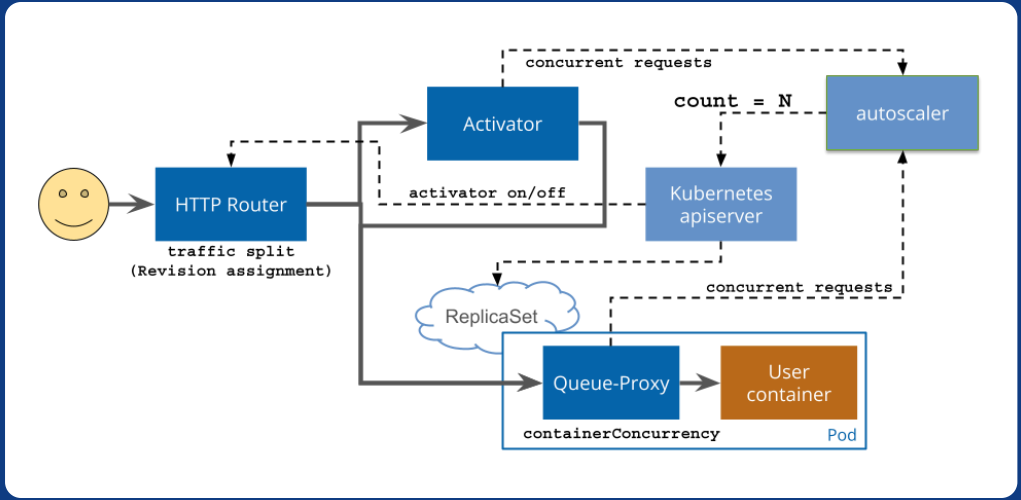
Before



After



Knative — the request flow



- **Queue-proxy** sidecar counts requests in flight, enforces `containerConcurrency`, reports to the autoscaler
- **Activator** buffers requests when there are too few (or zero) Pods and pokes the autoscaler
- **Proxy mode** — activator in the path when `demand + target-burst-capacity > Pods × target`
- **Serve mode** — enough spare capacity, router → Pods directly
- `target-burst-capacity` default **200**; `-1` = activator always in the path

Diagram: knative.dev/docs/serving/request-flow (CC BY 4.0)

Knative — stable vs. panic window

- **Stable mode** — average concurrency over the **60 s** window
- **Panic window** = 10 % of it = **6 s**
- **Panic** when the 6 s window wants **≥ 200 %** of the ready Pods
- in panic: scale on the 6 s average, **never scale down**
- back to stable after a full stable window without the need

Setting	Default	Annotation
`stable-window`	60s	`…/window`
`panic-window-percentage`	10	`…/panic-window-percentage`
`panic-threshold-percentage`	200	`…/panic-threshold-percentage`

Global: ConfigMap `config-autoscaler` in `knative-serving`. Per revision: `autoscaling.knative.dev/*` annotations on the template.

Panic is Knative's answer to bursts: react within seconds, then hold the Pods. HPA's closest knob is `behavior.scaleUp`.

Round 2 · Go Demo App — setup

```
sem ← struct{}{}                    // one of WORKERS (10) slots
time.Sleep(rand(minWait..maxWait))  // I/O-like wait, no CPU
←sem
```

- 10 requests in flight per Pod, the rest **queue** inside the app
- Knative, target 10, min 1, max 10

```
# rounds/round2/ksvc.yaml
annotations:
  autoscaling.knative.dev/class: kpa.autoscaling.knative.dev
  autoscaling.knative.dev/metric: concurrency
  # requests in flight per Pod the autoscaler aims for
  autoscaling.knative.dev/target: "10"
  autoscaling.knative.dev/window: "60s"
  autoscaling.knative.dev/min-scale: "1"
  autoscaling.knative.dev/max-scale: "10"
```

Think: a DB connection pool, a thread pool, a downstream API. The app is **full** long before the CPU is busy.

Round 2 · Go Demo App — expectations

- What would a CPU HPA do here?
- `fast`: 50 VUs × ~55 ms — how many Pods?
- `slow`: 5 VUs × 1–3 s — queue or not?
- Where do the requests wait?

KubeCon CloudNativeCon North America 2026

🎬 Demo time

Round 2 · Go Demo App

Argo CD `challenge-round2` → `bench-all` → **Create Job** · Grafana `namespace=round2`



Round 2 · Go Demo App — results

Knative details



Round 2 · Go Demo App — results

Why 8 Pods when we have 50 parallel requests?



Round 2 · Go Demo App — how to improve

- `containerConcurrency: 10` — the queue moves from the app to the activator, which can route to new Pods
- `target-utilization-percentage` — how early to add a Pod
- `WORKERS` itself — more slots per Pod, fewer Pods?
- `min-scale` for the known baseline
- watch *Busy workers vs. capacity* and *Queued requests*, not CPU

Round 2 · Go Demo App — improved, before vs. after

Before

60 s window



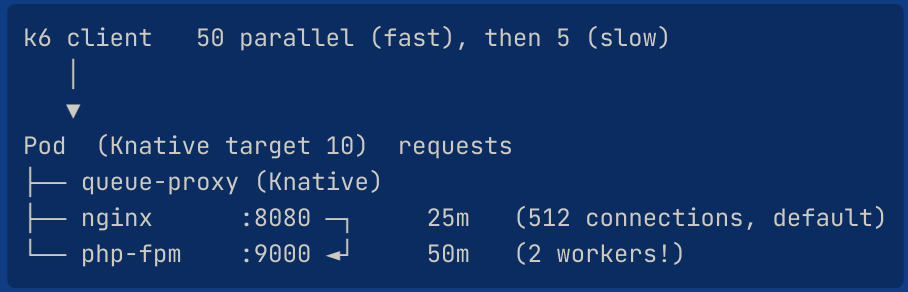
After

20 s window



Round 3 · nginx + php-fpm — setup

- **two app containers**, one scaling unit
- php-fpm `pm = static`, `FPM_WORKERS=2` — **on purpose** only 2
- extra requests wait in the php-fpm **listen backlog**
- Knative, target 10



Round 3 · nginx + php-fpm — expectations

- Which container is the bottleneck?
- What does one more replica cost?
- What would a CPU HPA average over?

🎬 Demo time

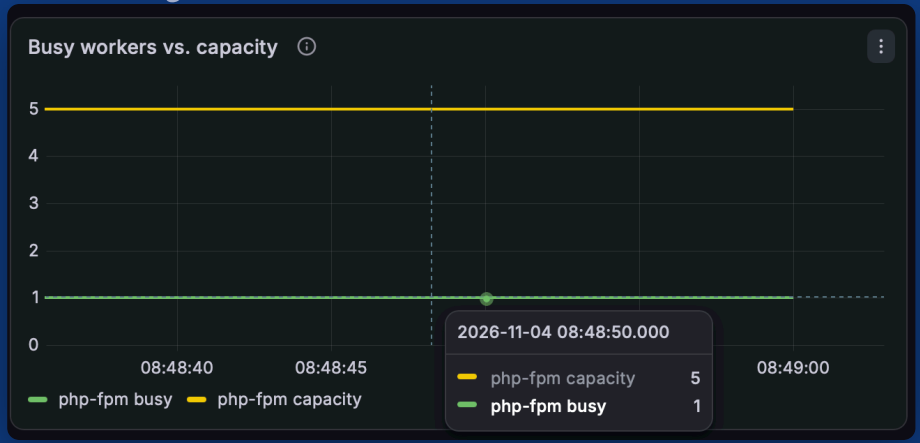
Round 3 · nginx + php-fpm

Argo CD `challenge-round3` → `bench-all` → **Create Job** · Grafana `namespace=round3`



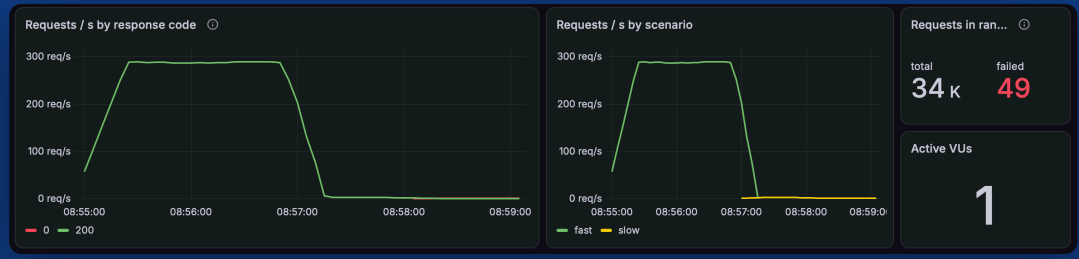
Round 3 · nginx + php-fpm — results

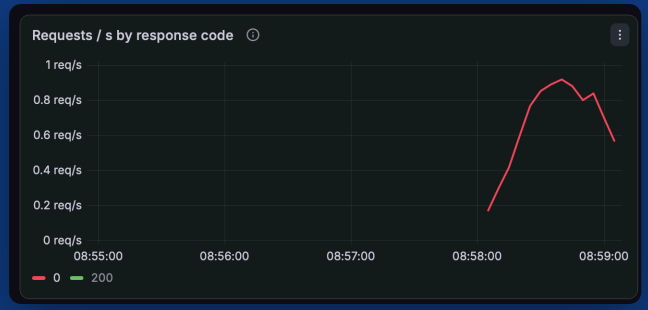
Why is 1 worker busy all the time?



Round 3 · nginx + php-fpm — results

Failed requests!



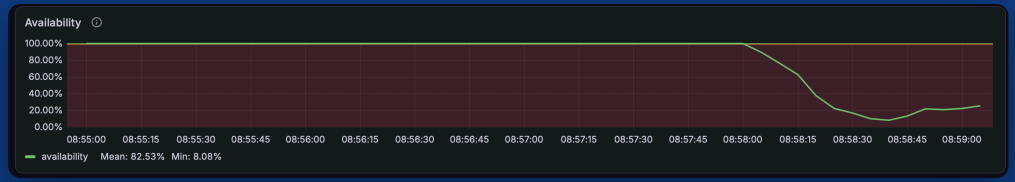


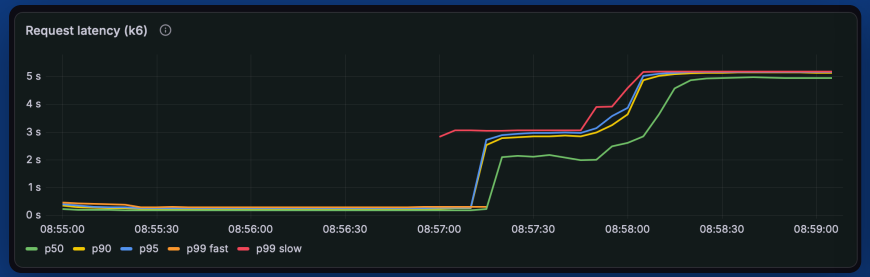
Status `0` = no response within the 5 s k6 timeout — during `slow`, only 5 parallel requests.

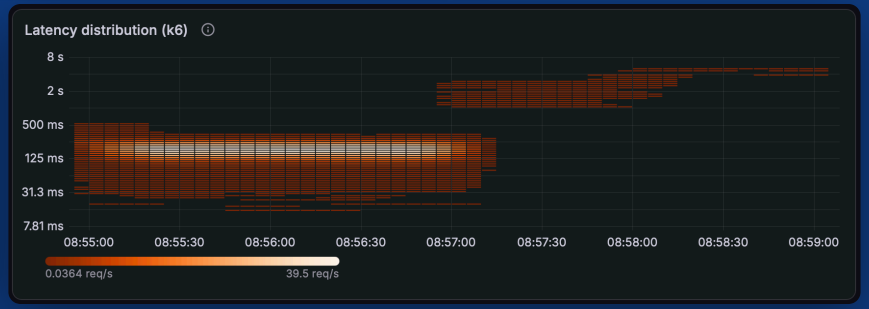
Round 3 · nginx + php-fpm — results

How to identify congestion

How "full" the system is







Round 3 · nginx + php-fpm — results

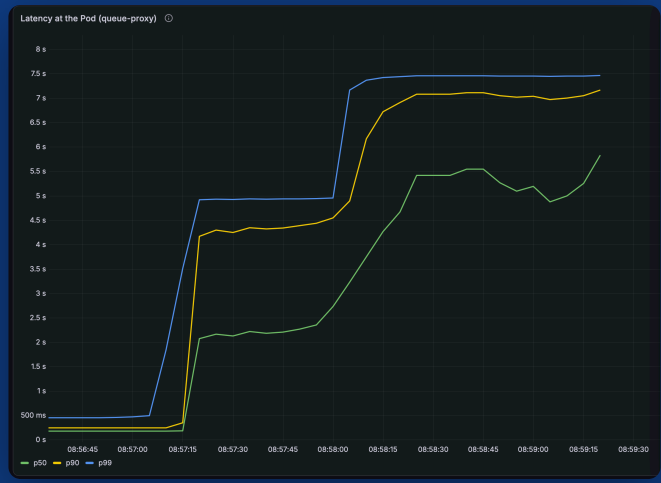
Which latency is correct?

Where do the requests go if the client times out?

Client k6



Server Knative queue-proxy



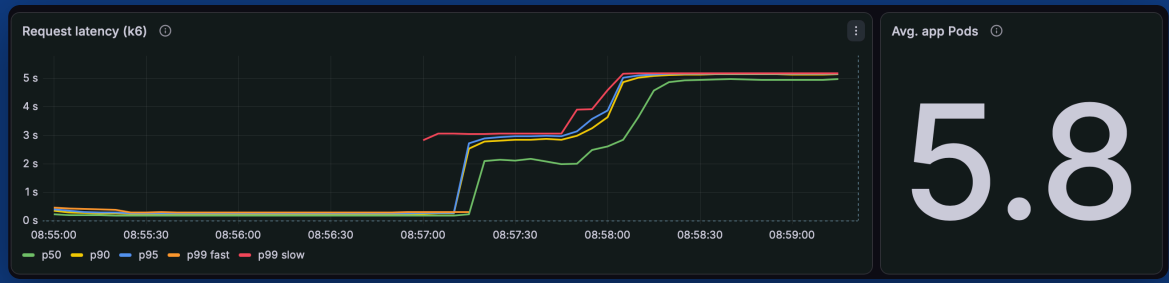
Round 3 · nginx + php-fpm — how to improve

- `target` = `FPM_WORKERS` — the worker pool is the capacity
- right-size **per container** — nginx needs little
- `pm = dynamic` vs. `static` — memory vs. readiness
- CPU HPA? → `ContainerResource` on `php-fpm`
- watch *php-fpm listen queue* and *busy vs. capacity*

Round 3 · Improved — before vs. after

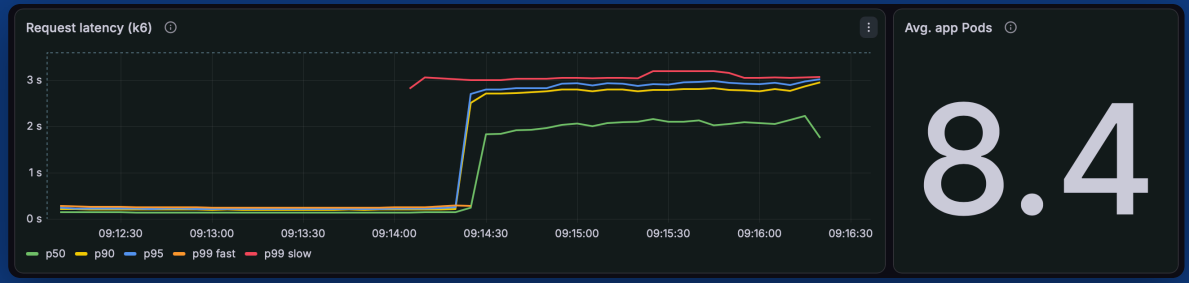
Before

target 10



After

target 2



Round 4 · Spring MVC — setup

- Spring MVC, `TOMCAT_THREADS=10` + accept queue
- JVM start under a **500m CPU limit**
- `STARTUP_DELAY_SECONDS=30` warm-up before readiness
- Knative, target 10

The usual enterprise app: slow JVM start, caches to fill, *Ready* only after warm-up.

Round 4 · Spring MVC — expectations

- How long from "scale up" to a Pod serving traffic?
- How much of the 120 s burst is left by then?
- Does panic mode help?

🎬 Demo time

Round 4 · Spring MVC

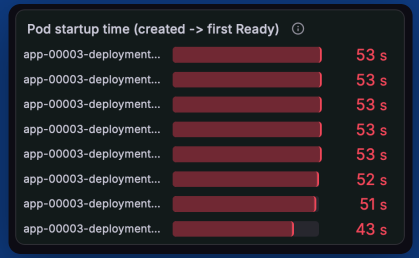
Argo CD `challenge-round4` → `bench-all` → **Create Job** · Grafana `namespace=round4`

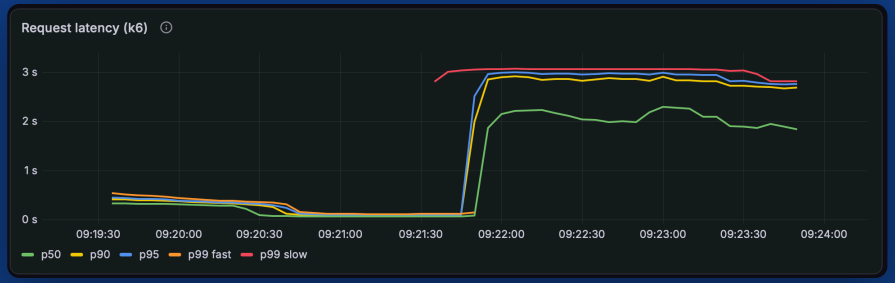


Round 4 · Spring MVC — results

What does slow startup mean?

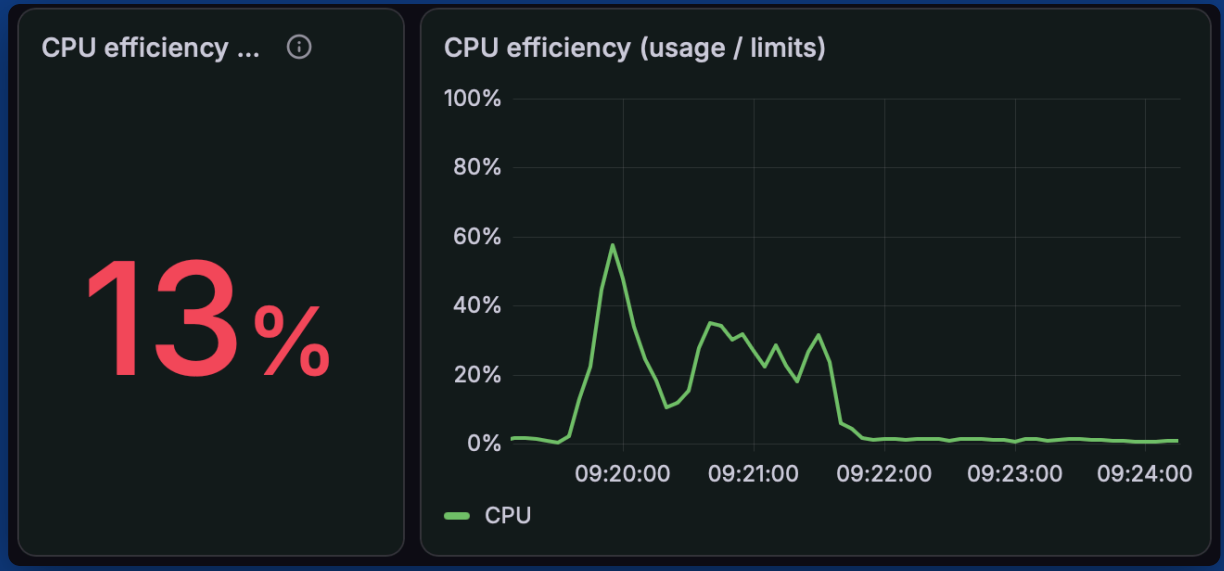






Round 4 · Spring MVC — results

Why that much CPU?



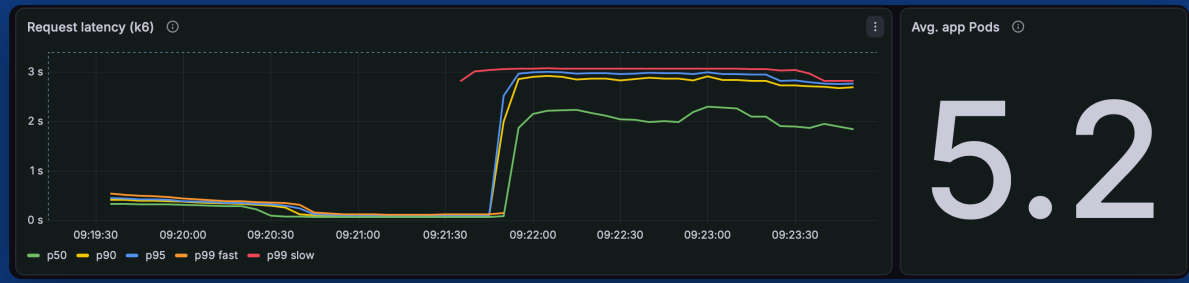
Round 4 · Spring MVC — how to improve

- `min-scale` — pay for headroom you can't create in time
- `initial-scale`, `scale-down-delay` — keep expensive Pods around
- lower `target` / utilization → scale **earlier**
- more CPU during startup (no limit, in-place resize)
- shorter warm-up — cache from a shared store, CDS / CRaC
- **GraalVM** native image — starts in milliseconds, no JIT warm-up
- startup time is part of your scaling budget — **measure it**

Round 4 · Improved? — before vs. after

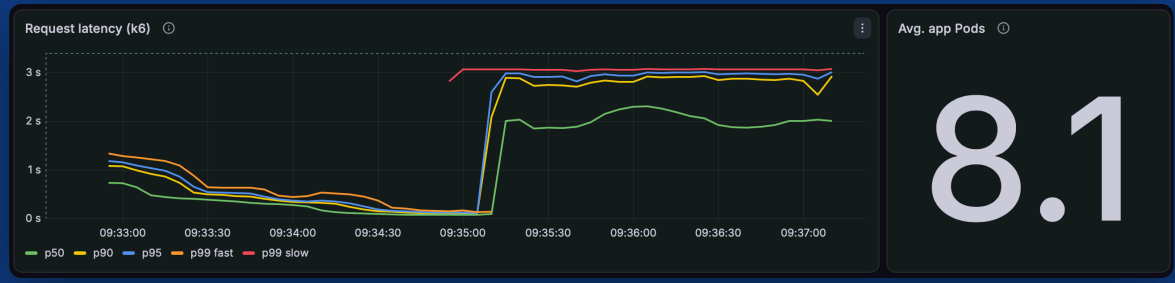
Before

utilization 70 %



Improved?

utilization 20 %



Round 5 · scale to zero — setup

- round 1 nginx again
- `min-scale: "0"` — idle ~60 s → **0 Pods**
- `imagePullPolicy: Always`
- next request waits in the **activator** for a cold start

```yaml
# rounds/round5/ksvc.yaml
autoscaling.knative.dev/min-scale: "0"
autoscaling.knative.dev/max-scale: "10"
...
image: nginx:1.29-alpine
imagePullPolicy: Always
```

A CPU HPA can't do this: no Pod = no CPU metric (`minReplicas: 0` needs the alpha `HPAScaleToZero` gate).

Round 5 · scale to zero — expectations

- How long does the first request wait?
- How many requests pay the cold start?
- What does it do to p99?
- Is one idle Pod worth it?

KubeCon CloudNativeCon North America 2026

🎬 Demo time

Round 5 · scale to zero

Argo CD `challenge-round5` → wait for 0 Pods → `bench-all` → **Create Job** · Grafana `namespace=round5`



Round 5 · scale to zero — results

Not bad, but…



Round 5 · scale to zero — how to improve

- `min-scale: "1"` — honestly, usually the answer
- `imagePullPolicy: IfNotPresent`, smaller images
- `scale-to-zero-pod-retention-period`, `scale-down-delay` — stay warm longer
- scale to zero where nobody waits: previews, batch, dev
- cherry on top — only after everything else works

Knative benefits from the queue vs. HPA — the activator holds the first request while a Pod starts. HPA has no queue: with 0 Pods the request just fails.

Round 6 · BFF (Backend-For-Frontend) → backend — setup

```
k6 ──► BFF (app, MODE=bff, 10 workers) ──► backend (10 workers, sleeps minWait..maxWait)
        TIMEOUT_MS=3000                      deadline: BACKEND_TIMEOUT_MS=2500
```

- both tiers = the round 2 Go app
- the BFF does **not** sleep itself — it forwards the wait to the backend
- two Knative Services, **autoscale independently**, target 10 each
- per-request override: `?timeout=` / `?backend_timeout=`

The BFF holds a worker while it waits for the backend. A stuck backend fills the BFF too.

Round 6 · BFF → backend — expectations

- Which tier saturates first?
- `slow` waits 1–3 s, backend deadline 2.5 s — what happens?
- Scale only the BFF — does it help?

🎬 Demo time

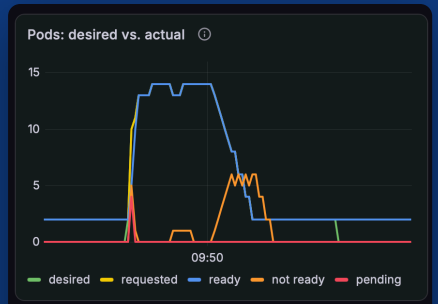
Round 6 · BFF → backend

Argo CD `challenge-round6` → `bench-all` → **Create Job** · Grafana `namespace=round6`

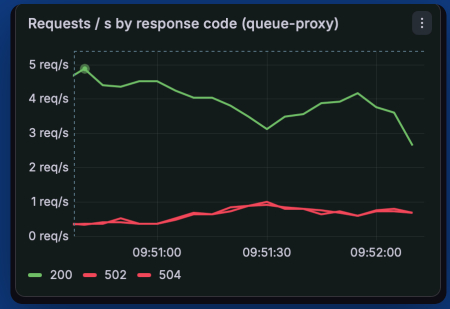


Round 6 · BFF → backend — results

Why do we have errors on slow requests?









Round 6 · BFF → backend — results

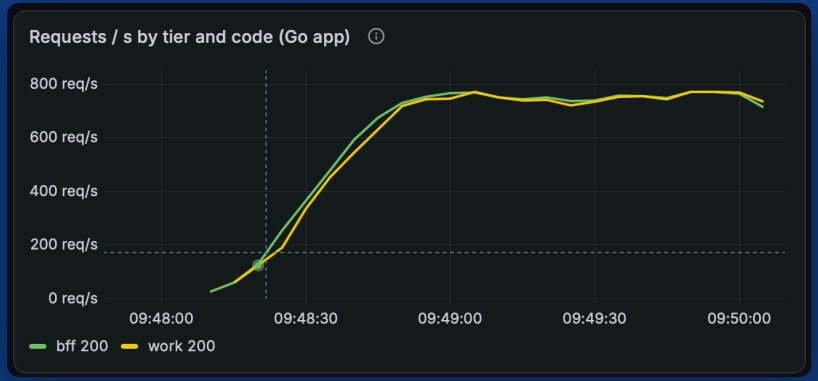
Twice the problem…



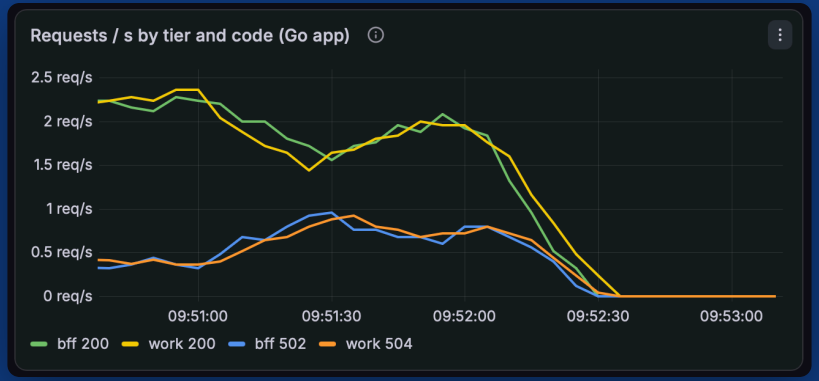
Round 6 · BFF → backend — results

Fast OK, slow fails

Fast 10–100 ms



Slow 1–3 s



- backend (`work`) deadline **2.5 s**, but `slow` waits up to 3 s → ~25 % of requests run out of time
- `504` **Gateway Timeout** — the backend hit its own 2.5 s deadline
- `502` **Bad Gateway** — the BFF got that `504` from upstream and passes it on (its own 3 s deadline never fires)

Round 6 · BFF → backend — how to improve

- timeouts **shrink along the chain**: client > BFF > backend
- timeouts follow the SLO, not the defaults
- **backend deadline 2.5 s → 3.1 s** — `slow` waits up to 3 s

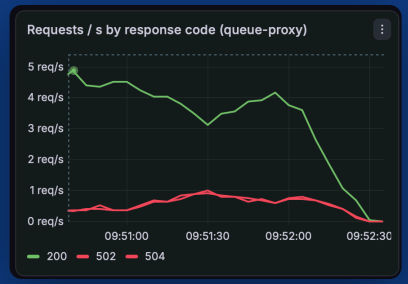
  quick fix — now backend 3.1 s > BFF 3 s, breaks the rule above
- propagate deadlines — don't work for callers who are gone (499)
- `target` and `WORKERS` per tier
- retries need a budget, or they multiply load
- watch *BFF → backend calls by result* (`timeout` vs. `504`)

Round 6 · Improved — before vs. after

Before

backend 2.5 s

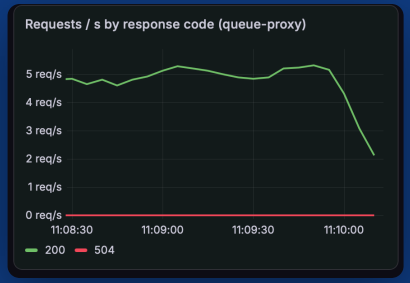




Improved

backend 3.1 s





But BFF queue + backend queue + 1–3 s work can still exceed the **5 s client timeout**

So autoscaling is easy, right?



Takeaways

1. **CPU is not throughput.** Scale on what saturates: workers, requests in flight, queue.
2. **Startup time is part of your scaling budget.** A Pod ready after 40 s misses a third of a 2-minute burst.
3. **Pods are scaling units, not containers.** Sidecars and co-located processes change the math.
4. **Timeouts and autoscaling are one design.** Measure the whole call chain.

KubeCon CloudNativeCon North America 2026

Thank you 📈

Questions? Find me in the hallway.

Repo + slides