ingress-nginx Hard Cases: Auth, Rate Limits, gRPC, Snippets

2026-07-18

A real OVH MKS lab migrates one nasty sample app from ingress-nginx to Envoy Gateway and HAProxy Unified Gateway: ten hard cases, real bugs, real gaps.

Every earlier part of this series was research: read the source, read the docs, verify a claim against a repo. This one is different. A real OVH Managed Kubernetes Service (MKS) cluster, a real domain, one deliberately nasty sample application, and the exact same 10-request test script run against ingress-nginx, then Envoy Gateway, then HAProxy Unified Gateway. Every result below is a real HTTP response or a real line read out of a running pod, not a docs claim.

► **TL;DR**

A single-node OVH MKS cluster ran all three targets, one at a time, against the same 10 hard cases: TLS, regex rewrite, canary, external auth, rate limit, WebSocket, gRPC, a large body, a tight timeout, and one snippet. Six cases behaved identically everywhere. Four did not: rate limiting diverges three different ways across the three targets (ingress-nginx's own 5x burst multiplier and `503`s, Envoy Gateway's stricter token bucket and `429`s, HAProxy Unified Gateway's complete lack of enforcement); external auth shows the same no-enforcement gap on HAProxy Unified Gateway; gRPC has no equivalent there at all (no `GRPCRoute` kind exists in that controller); and ingress-nginx's default `Strict-Transport-Security` header silently disappears on both Gateway API targets unless you re-add it (a trivial fix, but easy to forget), something none of the 10 hard cases were even designed to ask about. The single most structurally interesting finding: HAProxy Unified Gateway needs **one `HTTPRoute` per host**, not one per hard case — a real, load-bearing constraint found by reading a generated HAProxy config file inside a running pod, not guessed at. Every version pinned below is exactly what was actually installed on 2026-07-17/18 — expect specific bugs and gaps mentioned here to move as each project ships new releases.

The lab

The cluster is a real, single-node OVH MKS instance (Kubernetes v1.35.2, 4 vCPU / 16GiB), started from a clean slate — no Gateway API CRDs, no `IngressClass`, nothing but the default `kube-system` workloads. TLS uses real Let's Encrypt **staging** certificates via cert-manager's DNS-01 challenge against a real domain, issued through this blog's own cert-manager-webhook-libdns. The real lab domain has been replaced by the placeholder `hard-case.example` (RFC 2606's reserved documentation TLD) everywhere in this article and in the companion repo — the two hostnames

that matter are `lab.hard-case.example` for every HTTP/HTTPS case and `grpc.lab.hard-case.example` for gRPC, which needs its own host since gRPC method paths are fixed by the `.proto` file and can't be usefully prefix-matched the way plain HTTP paths can.

The sample app is `mccutchen/go-httpbin` (an actively maintained, official multi-arch image — no fork, no custom build) plus `moul/grpcbin` for the gRPC case, run as two versions (`go-httpbin-v1`/`-v2`) so the canary case has something real to split traffic between. All ten hard cases and their exact manifests live in the companion repo, `ingress-nginx-migration-lab`, on Codeberg — every command, every YAML file, every raw result below is reproducible from that repo directly, not reconstructed from notes.

Each target ran the identical test script (`tests/run-tests.sh`) against whatever was live at the time, with results committed as timestamped Markdown files under `tests/results/`. Three phases:

1. **Baseline** — ingress-nginx, Helm chart v4.15.1, the current stable release before this lab makes any changes at all.
2. **Envoy Gateway v1.8.2** — installed via its own `gateway-helm` chart, which bundles both the upstream Gateway API CRDs and Envoy Gateway's own CRDs in one install.
3. **HAProxy Unified Gateway ("HUG") v1.0.5** — installed from plain manifests (no Helm chart in this project's public repo), the latest stable tag.

i Why HUG

HAProxy Unified Gateway is a materially younger project than Envoy Gateway, confirmed by reading its actual source rather than assuming from its README: at v1.0.5 it implements only `HTTPRoute` from the Gateway API — no `GRPCRoute` kind exists in the controller at all — and it has no dedicated rate-limit or external-auth policy CRD (only low-level, raw-HAProxy-config primitives like stick-table ACLs and an `auth_realm` field for HTTP Basic/Digest auth, not a portable Gateway API policy attachment). Two decisions were made up front, before any live test, to keep the comparison honest rather than force a result: rate limit and external auth are documented as unsupported with no live attempt, and gRPC is documented as a flat capability gap with no live attempt, since there's no `GRPCRoute` kind to even try. That leaves **6 of 10 hard cases** actually tested against HUG: TLS, canary, WebSocket, regex rewrite, large body, and tight timeout.

Results at a glance

Hard case	ingress-nginx	Envoy Gateway v1.8.2	HAProxy Unified Gateway v1.0.5	Verdict
TLS	`200`	`200`	`200`	Equivalent
Regex rewrite	rewritten path served	rewritten path served	rewritten path served	Equivalent
Canary (20 req.)	11/9 split	10/10 split	9/11 split	Equivalent
WebSocket echo	round-trip OK	round-trip OK	round-trip OK	Equivalent
Large body (5MB)	`200`	`200`	`200`	Equivalent (no size cap anywhere)
Tight timeout	`504`	`504`	`504`	Equivalent — but see findings below
External auth	`401`/`200` (enforced)	`401`/`200` (enforced)	`404` (not enforced)	**Real gap (HUG)**
Rate limit (20 req.)	11/9 `200`/`503`	2/18 `200`/`429`	`404` ×20 (not enforced)	**Real divergence (all three differ)**
gRPC	5 services listed	5 services listed	not attempted — no `GRPCRoute` kind	**Capability gap (HUG)**
Snippet	`200` + custom header	`404` — no route	`404` — no route	Expected gap (by design, both targets)

Last verified: 2026-07-17/18 — Kubernetes v1.35.2, ingress-nginx Helm chart v4.15.1, Envoy Gateway v1.8.2, HAProxy Unified Gateway v1.0.5.

Walking through the hard cases

TLS

All three targets terminate TLS with the same cert-manager-issued Secret and serve `200` on `GET /get`. ingress-nginx's `Ingress` just lists the host and `secretName`:

```
spec:
  tls:
    - hosts:
        - lab.hard-case.example
      secretName: hard-case-lab-tls
```

Both Gateway API targets reuse the exact same Secret through a listener's `tls.certificateRefs` — no new certificate issuance needed for either migration:

```
listeners:
  - name: https
    port: 443    # 31443 on HAProxy Unified Gateway — see the fixed-port finding below
    protocol: HTTPS
    tls:
      mode: Terminate
```

```
      certificateRefs:
        - name: hard-case-lab-tls
```

No RCA needed — clean equivalence on all three.

Regex rewrite

ingress-nginx's `use-regex` + a capture-group `rewrite-target` sends `/rewrite/foo` to `/anything/foo`:

```
annotations:
  nginx.ingress.kubernetes.io/use-regex: "true"
  nginx.ingress.kubernetes.io/rewrite-target: /anything/$1
spec:
  rules:
    - http:
        paths:
          - path: /rewrite/(.*)
            pathType: ImplementationSpecific
```

Core Gateway API's `URLRewrite` filter only supports prefix replacement, not arbitrary regex capture groups — narrower in general (Part 3 already flagged capture-group rewrites as a red case for core Gateway API), but sufficient for this specific hard case since the captured suffix carries through unchanged either way:

```
rules:
  - matches:
      - path: { type: PathPrefix, value: /rewrite }
    filters:
      - type: URLRewrite
        urlRewrite:
          path: { type: ReplacePrefixMatch, replacePrefixMatch: /
anything }
```

Identical on both Envoy Gateway and HUG. No RCA needed — this specific case's shape happens to sit entirely inside what `ReplacePrefixMatch` can do.

Canary

ingress-nginx's canary model needs a **second** `Ingress` object sharing the same host and path, marked `canary: "true"` — the weight isn't a field on the primary object at all:

```
annotations:
  nginx.ingress.kubernetes.io/canary: "true"
  nginx.ingress.kubernetes.io/canary-weight: "50"
```

Gateway API doesn't need the second-object split — weighted `backendRefs` on one `HTTPRoute` cover both the primary and canary backend directly, a genuine portability win over ingress-nginx's own mechanism:

```
backendRefs:
  - name: go-httpbin
    port: 80
    weight: 50
  - name: go-httpbin-canary
    port: 80
    weight: 50
```

Identical on HUG too (it has its own dedicated `blue-green` example proving the same mechanism). Split ratios across a 20-request sample (11/9, 10/10, 9/11) are all within normal sampling noise of a 50/50 weight — no RCA needed.

WebSocket

No annotation on ingress-nginx at all — it proxies `Upgrade`/`Connection` headers by default. No special configuration on either Gateway API target either; both proxy the upgrade the same way. Echo round-trip (`hard-case-ws-probe` sent and received back intact) confirmed on all three. Worth naming as an operational aside rather than a controller finding: the test tooling itself needed three attempts before finding a WebSocket client image that worked correctly against TLS (see "Test methodology" below) — the controllers were never the problem here. What finally worked was `docker.io/mtilson/websocat`, a community fork built specifically to fix this exact TLS crash in the original `vi0oss/websocat` image.

Large body (5MB POST)

No explicit size cap found on any of the three targets during this test — ingress-nginx's `proxy-body-size: 25m` annotation is set generously above the 5MB test payload rather than removed entirely (a limit was configured, just not tight enough to trigger), and neither Envoy Gateway's `ClientTrafficPolicy` nor HUG's `Backend`/`Defaults` CRDs were configured with any equivalent cap for this lab. `200` on all three. Not a finding about equivalence — an open question this specific lab run didn't push hard enough to answer, carried over unchanged from Part 4's own note on the same gap.

Tight timeout

Symptom: all three targets return `504` for a request whose backend (`/delay/5`) deliberately runs longer than the configured proxy timeout (3s) — on the surface, a clean pass everywhere.

Evidence: ingress-nginx's `proxy-read-timeout: "3"` is a plain annotation. Both Gateway API targets use the identical core field, `rules[].timeouts.backendRequest: 3s`:

```
rules:
  - matches:
      - path: { type: PathPrefix, value: /slow }
    filters:
      - type: URLRewrite
        urlRewrite: { path: { type: ReplacePrefixMatch,
replacePrefixMatch: /delay/5 } }
```

```
    timeouts:
      backendRequest: 3s
```

Root cause of why this is a real finding, not a boring pass: Part 4's own empirical testing of Envoy Gateway found an overwrite bug on this exact field — `BackendRequest` unconditionally overwriting `Request` without a `Retry` block, producing wrong effective timeouts. Separately, this lab's own Phase C planning stage predicted, from a source grep that found no "BackendRequest" string in HUG's route-translation code, that the field probably wasn't handled by HUG at all.

Fix or accepted limitation: neither pessimistic prediction held up live. Envoy Gateway v1.8.2 returns a clean `504` with no sign of the Part 4 overwrite bug — it appears fixed since that testing. HUG's case needed direct evidence, not a repeat grep: reading the actual generated `/usr/local/hug/haproxy.cfg` inside the running pod showed `timeout server 3000` on the timeout route's backend, exactly matching the configured `3s` — the field is correctly translated, the earlier grep had simply missed the actual code path.

Migration lesson: a `grep` for a field's Go identifier finding nothing is weaker evidence than reading the live, generated configuration — prefer the latter whenever the two disagree, and don't present a grep-based absence as confirmed non-support without a live check to back it up.

External auth

Symptom: ingress-nginx and Envoy Gateway both enforce correctly — `401` without a bearer token, `200` with one. HAProxy Unified Gateway returns `404` in both cases.

Evidence: ingress-nginx's `auth-url` points at go-httpbin's own `/bearer` endpoint as the subrequest target:

```
annotations:
  nginx.ingress.kubernetes.io/auth-url: "http://go-httpbin.hard-case-lab.
svc.cluster.local/bearer"
```

Envoy Gateway's `SecurityPolicy` does the same thing through `extAuth.http`, with one real correction needed during setup: the plain `path` field *prepends* to the original request path (producing `/auth/protected`, not what's wanted), so `pathOverride` — which replaces the path outright — is required instead, confirmed from the field's own doc comment before it was ever tested live:

```
extAuth:
  http:
    backendRefs:
      - name: go-httpbin
        port: 80
    pathOverride: /bearer
```

HUG has no such CRD at all — its only auth-adjacent primitive is `auth_realm` on the `Backend`/`Defaults` CRDs, which implements HTTP Basic/Digest auth, not delegation to an external service.

Root cause: the `404` is coincidental, not a stricter or looser enforcement decision. The request reaches go-httpbin directly through HUG's catch-all route, unauthenticated — go-httpbin itself

simply has no `/protected` route of its own, so its own router returns Go's stdlib default "404 page not found" text. There is no enforcement happening at all.

Fix or accepted limitation: documented capability gap, confirmed via source before the live test (no external-auth CRD exists in the controller), and confirmed live (the exact 404 text proves the request wasn't intercepted by any policy layer at all).

Migration lesson: "the response code looks like a failure" and "the request was rejected by a security policy" are two different claims — always check *why* a response looks the way it does before treating a status code as proof of enforcement, especially when a capability is already known to be absent going in.

Rate limit

Symptom: three different behaviors across three targets, none of them the same.

Evidence: 20 concurrent requests against each target, tallied by status code:

- ingress-nginx (`limit-rps: "2"`): **11× `200`, 9× `503`**.
- Envoy Gateway (`BackendTrafficPolicy`, `rateLimit.local.rules[].limit: {requests: 2, unit: Second}`): **2× `200`, 18× `429`**.
- HAProxy Unified Gateway: **20× `404`** — the same no-enforcement passthrough as the external-auth case above, not a rate-limit decision at all.

```
# Envoy Gateway's BackendTrafficPolicy — same nominal limit as ingress-
nginx's limit-rps: "2"
rateLimit:
  type: Local
  local:
    rules:
      - limit:
          requests: 2
          unit: Second
```

Root cause: ingress-nginx applies its own default 5x burst multiplier on top of the configured rate — a burst of 10 against 20 concurrent requests lets roughly half through. Envoy Gateway's local rate limit is a plain token bucket with no documented implicit burst at all — a much stricter enforcement of the same nominal "2 req/s" number. HUG has no rate-limit policy CRD, so the `404` is the identical catch-all-passthrough finding as external auth, not a third distinct enforcement style.

Fix or accepted limitation: none needed for ingress-nginx or Envoy Gateway — both enforce *something*, just at different strictness and with different HTTP semantics (`503` reads as a generic server-side failure; `429` explicitly signals "you were rate-limited, retry"). HUG's gap is the same accepted, documented capability limitation as external auth.

Migration lesson: "same nominal rate-limit number" does not mean "same client-visible behavior." A client that treats `503` and `429` differently (different retry logic, different alerting) will behave differently after this specific migration even though the configured number didn't change — this

is exactly the kind of silent behavioral drift Part 3's annotation matrix exists to warn about, now confirmed live instead of just predicted.

gRPC

Symptom: ingress-nginx and Envoy Gateway both correctly proxy gRPC reflection, listing the same 5 services. HAProxy Unified Gateway was never actually tested — and one line in this lab's raw Phase C result file looks like a HUG result but isn't.

Evidence: ingress-nginx uses `backend-protocol: GRPC` on its own host (gRPC method paths are fixed by the `.proto`, so they can't be prefix-matched the way HTTP paths can); Envoy Gateway uses the distinct `GRPCRoute` kind on its own listener. Both list:

```
addsvc.Add
grpc.gateway.examples.examplepb.ABitOfEverythingService
grpc.reflection.v1alpha.ServerReflection
grpcbin.GRPCBin
hello.HelloService
```

Root cause of the stray result line: `grpc.lab.hard-case.example` still resolved, via real DNS untouched since the baseline phase, to ingress-nginx's own LoadBalancer IP when the HUG test run happened — because gRPC correctly wasn't attempted against HUG (per the scope decision above), the test script's `GRPC_HOST_IP` override was correctly left unset, and `grpcurl` silently fell through to real DNS and tested the *still-live baseline* instead of HUG.

Fix or accepted limitation: documented capability gap — no `GRPCRoute` kind exists anywhere in HUG's controller code, confirmed from source before any test was attempted. The stray result line is struck from this lab's findings, not counted as evidence either way.

Migration lesson: when a target is deliberately excluded from one test in a shared test harness, double-check that the *harness itself* doesn't quietly fall back to a still-live peer target's endpoint through DNS or default routing — an intentional gap and an accidental one can produce output that looks identical unless the underlying network path is checked, not just the response body.

Snippet

ingress-nginx's `configuration-snippet` injects an arbitrary raw NGINX directive — real custom proxy code, not configuration, requiring `allow-snippet-annotations` to even be enabled (a real security trade-off this lab makes deliberately, on a controller that ships this gate off by default):

```
annotations:
  nginx.ingress.kubernetes.io/configuration-snippet: |
    add_header X-Lab-Snippet "nginx-only-no-portable-equivalent" always;
```

`200` plus the custom header on ingress-nginx; `404` — no route exists at all — on both Envoy Gateway and HUG, since neither has anything resembling a snippet mechanism and no route was ever created to intentionally exercise the gap. This is the same "no portable equivalent by design" finding Part 3 already named for every migration target evaluated in this series, now surfaced more starkly than a matrix cell can: the whole path is unreachable, not just one custom header missing.

Gotchas that aren't about any one hard case

Five real findings surfaced during setup and during header-level diffing, not during any single hard-case test — each still gets the same RCA treatment, because each one would bite a real migration exactly the same way.

The default HSTS header disappears on both Gateway API targets

Symptom: none of the 10 hard cases asked about response headers beyond the one custom header in the snippet case — but diffing the *full* response headers across all three targets, not just status codes, surfaced a security-relevant difference nothing was testing for.

Evidence: ingress-nginx's TLS response carries `strict-transport-security: max-age=31536000; includeSubDomains` on every HTTPS response. Neither Envoy Gateway nor HAProxy Unified Gateway sends any `Strict-Transport-Security` header at all — the same `GET /get` over HTTPS comes back without it on both.

Root cause: ingress-nginx enables HSTS by default in its own global configuration; the two Gateway API targets make no such default policy decision. This is a difference in default posture, not a capability gap — both targets implement the core Gateway API `ResponseHeaderModifier` filter (confirmed from source at the installed versions: Envoy Gateway v1.8.2 ships a dedicated response-header task, and HAProxy Unified Gateway v1.0.5 handles `HTTPRouteFilterResponseHeaderModifier` in `k8s/gate/haproxy/filters/http_filters.go`, with its own integration test for it). Adding HSTS back is a two-line filter on the same `HTTPRoute`, on either target:

```yaml
filters:
  - type: ResponseHeaderModifier
    responseHeaderModifier:
      add:
        - name: Strict-Transport-Security
          value: "max-age=31536000; includeSubDomains"
```

Fix or accepted limitation: not a bug, and not hard to fix — but easy to *forget*. The header is opt-in on both Gateway API targets and opt-out (on by default) on ingress-nginx, so a migration that doesn't explicitly re-add it silently drops a header real browsers act on, with no error, no failed test, and nothing in a status-code-only comparison to catch it.

Migration lesson: diff full response headers before and after a migration, not just status codes and bodies. The most consequential difference in this entire lab was a header that no hard case was designed to test, and the fix is trivial once you know to look — it only showed up because the comparison looked at everything the response carried, not just whether the request succeeded.

OVH's PROXY-protocol annotation doesn't reach Envoy Gateway's Service

Symptom: Envoy Gateway's auto-created `LoadBalancer` Service was missing the OVH MKS PROXY-protocol annotation (`loadbalancer.openstack.org/proxy-protocol: "v2"`) that preserves the real client IP at the load balancer.

Evidence: `kubectl get svc ... -o jsonpath='{.metadata.annotations}'` showed no such annotation after the first apply, even though Gateway API's own `Gateway.spec.infrastructure.annotations` field had been set with exactly that intent.

Root cause: `spec.infrastructure.annotations` is documented but is not propagated to the auto-created Service by Envoy Gateway — a real portability gap, not a misconfiguration.

Fix: Envoy Gateway's own `EnvoyProxy` CRD, referenced from the `Gateway` via `spec.infrastructure.parametersRef`:

```
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: EnvoyProxy
metadata:
  name: eg-proxy-config
  namespace: hard-case-lab   # must match the Gateway's own namespace — parametersRef is
                             # local-only, confirmed from gateway-api's own type definitions
spec:
  provider:
    type: Kubernetes
    kubernetes:
      envoyService:
        annotations:
          loadbalancer.openstack.org/proxy-protocol: "v2"
```

That annotation only makes the load balancer *send* a PROXY protocol preamble — Envoy's own listener doesn't parse it unless separately told to via a `ClientTrafficPolicy`, whose own doc comment states plainly that "Proxy Protocol must be present when this field is set, else the connection is closed." The two sides of this contract were wired up together, before any test traffic was sent, not discovered as an outage afterward. The specific field name shifted between versions too: this repo's current main branch has a structured `proxyProtocol.version` field, but the actual installed **v1.8.2** release only has the older `enableProxyProtocol` boolean — confirmed by checking the source out at the exact installed tag (`git show v1.8.2:api/v1alpha1/clienttrafficpolicy_types.go`) after a live "unknown field" rejection, not by trusting the local clone's default branch.

Migration lesson: PROXY-protocol propagation is always a two-sided contract — the load balancer must send it and the proxy must be told to expect it — and when checking a project's source for a specific field name or struct, check it out at the exact tag actually installed, not whatever the local clone's working tree happens to be on.

HAProxy Unified Gateway's own CRDs aren't in the standard install

Symptom: the first `kubectl apply -k` against HUG's manifests created everything except its `Global` and `HugConf` custom resources, failing with "no matches for kind … ensure CRDs are installed first."

Root cause: HUG's own CRDs (`Global`, `HugConf`, `Defaults`, `Backend`, `HugGate`, under `gate.v3.haproxy.org`) are separate from the standard Gateway API CRDs and aren't installed by any manifest in the project's own `example/deploy/hug/` — they need a one-shot Job (`hug --job-check-crd`) with its own, more privileged ServiceAccount, deliberately kept separate from the running controller's own more limited RBAC.

Fix: run that Job once, before applying anything else. HUG's install docs also offer a companion job (`--job-gwapi=1.5.0`) that installs the standard Gateway API CRDs — deliberately **not** run in this lab, since Envoy Gateway's own `gateway-helm` chart had already installed a newer set and HUG only needs `HTTPRoute`, already present; running a second, possibly different-versioned Gateway API CRD install risked disturbing what the still-live Envoy Gateway phase depended on, for no benefit.

Migration lesson: "installed the Gateway API CRDs" and "installed this controller's own custom resources" are two separate steps for at least one real project in this space — check for a controller-specific CRD-install mechanism explicitly, rather than assuming one Helm/manifest install covers everything a quickstart doc implies.

HAProxy Unified Gateway needs one `HTTPRoute` per host, not per hard case

Symptom: after fixing the CRD-ordering issue above, every path except one returned a plain `404` — including paths that should have matched a route that was demonstrably applied and `Accepted`.

Evidence: `kubectl exec` into the running controller pod and reading its generated HAProxy config directly (not inferring from the symptom) showed the actual problem: `/usr/local/hug/maps/hug_https_31443/listener_route_exact_match.map` had the identical key (`hard-case-lab/hug_https/lab.hard-case.example`) repeated four times, once per `HTTPRoute` object, each with a different value.

Root cause: HUG selects which `HTTPRoute` serves a request via a HAProxy `map()` lookup keyed by `(listener, host)` — not `(listener, host, path)`. With four separate `HTTPRoute` objects all on the same host, the generated map has four entries under one identical key, and HAProxy's `map()` converter on a duplicate key returns only the first-declared value. Every path belonging to any of the other three routes 404s from `backend_not_found`, regardless of what path was actually requested. (This symptom briefly raised the question of whether ingress-nginx and Envoy Gateway, still running on separate LoadBalancer IPs alongside HUG, might be colliding somehow — ruled out immediately: each Gateway API controller filters strictly by its own `controllerName`, and all three targets have entirely separate LoadBalancer IPs. The cause was entirely internal to HUG's own per-host route-selection map.)

Fix: consolidate every attempted hard case into **one** `HTTPRoute` per host, with multiple `rules[]` inside it, instead of one `HTTPRoute` object per hard case — the pattern that worked without issue on both ingress-nginx (separate `Ingress` objects, a different mechanism entirely) and Envoy Gateway (which has no problem with multiple `HTTPRoute` objects on one host).

Migration lesson: "one Gateway API resource per hard case" is a style choice that happened to work on two of the three targets in this lab — not a Gateway API guarantee. At least one

real implementation imposes a structural one-route-per-host constraint underneath a spec that otherwise allows either shape.

HAProxy Unified Gateway's Service reconciles back to its own fixed ports

Symptom: the controller's `Service`, explicitly configured to map external ports `80`/`443` to the container's `31080`/`31443`, silently reverted to serving on `31080`/`31443` directly.

Evidence: the Service's own `last-applied-configuration` annotation still showed the intended `80→31080`/`443→31443` mapping, but the live spec had been renamed (`http-31080`/`https-31443`) and re-pointed to ports `31080`/`31443` literally — confirmed by watching the two diverge after the controller's own reconciliation loop ran, then confirmed live: port `443` on this load balancer times out entirely, `31443` serves traffic.

Root cause: HUG's controller actively reconciles its managed Service back to its own fixed container ports — there is no per-Gateway auto-provisioned LoadBalancer the way Envoy Gateway has; one shared data-plane process serves every `Gateway` object on those same two fixed ports.

Fix or accepted limitation: accepted as a real architectural difference, not something to fight — the test script gained a `HOST_PORT` override (default `443`, set to `31443` for this phase) rather than trying to force the Service onto standard ports.

Migration lesson: confirm which port a target's load balancer actually serves traffic on before trusting a manifest's stated intent — for at least one real implementation, the controller's own reconciliation loop overrides the manifest silently, with no error or event surfaced anywhere.

Test methodology, not controller bugs

Four bugs surfaced in the test script itself during the very first baseline run against ingress-nginx — worth naming so none of them get misattributed to a controller, but kept brief since none are migration findings: the canary tally was meaningless until go-httpbin's `USE_REAL_HOSTNAME` env var was set (otherwise every pod reports an identical dummy hostname); the rate-limit test needed 20 *concurrent* requests, not sequential ones, to ever trigger a limit at all; the large-body test's first `400` was curl's own default `Content-Type` for `--data-binary` tripping go-httpbin's form-decoding, fixed with an explicit `Content-Type: application/octet-stream` header; and the WebSocket test needed a third container image (`docker.io/mtilson/websocat`) before finding one that didn't either not exist or crash on TLS. All four were fixed once, in the shared test script, before any comparison run — every phase after the baseline reused the corrected logic unchanged.

What this lab does and doesn't prove

This is one real migration, not a statistically representative one. A single-node cluster, Let's Encrypt **staging** certificates that were never promoted to production, and no real production load or concurrency behind any of these results — the rate-limit and canary numbers above are real observed counts, not numbers from a load-testing tool built for statistical confidence. Only 2 of the 8 candidates from Part 4's matrix got a live migration here, chosen for the sharpest contrast between two Gateway API implementations at different maturity levels, not because the other 6 wouldn't be worth testing too. And go-httpbin itself never got a security review as a production application — it's a well-known, purpose-built test target, not a stand-in for a real backend.

Sources

- `ingress-nginx-migration-lab` — the companion repo: every manifest, every command, every raw result file this article is drawn from
- envoyproxy/gateway — source, tag `v1.8.2`
- haproxytech/haproxy-unified-gateway — source, tag `v1.0.5`
- kubernetes-sigs/gateway-api — core API types referenced throughout (`HTTPRoute`, `GRPCRoute`, `Timeouts`, `URLRewrite`)
- mccutchen/go-httpbin — the sample application
- moul/grpcbin — the gRPC test backend
- Part 3 — The Matrix — the per-annotation findings this lab's results are checked against
- Part 4 — Replacement Candidates — source of the Envoy Gateway timeout-overwrite bug this lab confirms fixed at v1.8.2

What's next

This closes the empirical arc of this series — every part in the original plan is now published. The series hub still tracks publication status, but per its own stated intent, that doesn't mean the series is *finished*: Part 3's living annotation matrix, and any future target or annotation worth adding to it, can still grow independently of this lab's own scope. If you're migrating your own cluster off ingress-nginx, the companion repo is built to be re-run against your own targets, not just read.