ingress-nginx Annotation Compatibility Matrix

2026-07-18

Why classifying ingress-nginx annotations by name is dangerous: what migrates cleanly to Gateway API or another Ingress controller, and what silently breaks.

Part 2 built the inventory. This part answers the question that inventory exists to feed: for the annotations that actually turn out to be hard cases — not every annotation, but the ones where the answer isn't obvious — does migration go cleanly, or does it silently break? The honest answer changes row by row — and, in one case in this matrix, changes depending on the *value* of the annotation, not just its name.

► TL;DR

Classifying annotations by name alone is dangerous. `rewrite-target` is green for a simple prefix rewrite and red for a regex capture group — same annotation, opposite verdict. Some targets fail loudly (NGINX Gateway Fabric rejects the standard `Timeouts` field outright); Traefik fails silently on that same core field (accepted, zero effect) — though its separate ingress-nginx-compat provider gets the underlying annotations right through a different mechanism, so the real answer depends on which migration path you actually take. The full matrix — 17 rows so far, growing — lives as CSV in the companion repo; this part walks through the groups as per-section tables, with prose reserved for the findings sharp enough to need it.

⚠ Long, technical read

This part runs well over half an hour of reading time — deliberately so, since the whole point of a per-annotation matrix is the detail a shorter summary would flatten. It assumes familiarity with both `Ingress` and Gateway API. If you're short on time, the TL;DR above and the cover heatmap capture the shape of the findings; come back to the per-section tables when the specific annotation you depend on is actually in play.

Why classify by behaviour, not by name

`rewrite-target` is the cleanest illustration of why this matrix exists at all. A simple prefix or full-path rewrite maps directly onto Gateway API's core `URLRewrite` filter — confirmed straight from the Gateway API source (`apis/v1/httproute_types.go`), the `HTTPPathModifierType` enum is exactly `ReplaceFullPath` and `ReplacePrefixMatch`. Nothing else. The moment `rewrite-`

`target` uses a regex capture group — `rewrite-target: /$1` paired with a regex path — there is **no equivalent in core Gateway API at all**. Same annotation. Opposite verdict. An inventory that only records the annotation's name, not the value actually in use, would misclassify half of these.

There's a third twist worth naming here too: F5′s own NGINX Ingress Controller uses the identical annotation name, `nginx.org/rewrite-target`, and — confirmed from its `VirtualServerRoute` CRD docs — its `rewritePath` field explicitly supports capture groups (`$1` through `$9`). Migrating the regex case to the same vendor's own controller carries the capability over natively; migrating it to *core* Gateway API does not. Same annotation, same name even, and the verdict still depends on *where* you're going, not just what value you're using. Kong's own `konghq.com/rewrite` Ingress annotation carries the same capability over too — confirmed from source, it parses `$1`/`$2` capture syntax into Kong's own template format. HAProxy's own Kubernetes Ingress Controller carries it over too, through a third mechanism again — confirmed from source, its `path-rewrite` annotation takes a match regex and a replacement format string (`pkg/annotations/ingress/reqPathRewrite.go`), compiled straight into HAProxy's own native `replace-path` directive (`MatchRegex`/`ReplaceFmt` fields, `client-native/configuration/http_request_rule.go`) — the same regex-substitution mechanism HAProxy uses outside Kubernetes entirely. Traefik carries it over from a fourth direction — twice, in fact: its own ingress-nginx-compat provider auto-translates the annotation directly into a purpose-built `dynamic.RewriteTarget` type (with automatic `X-Forwarded-Prefix` handling the other three don't offer), and separately its classic `Middleware` CRD has a `ReplacePathRegex` field with real capture-group support, confirmed from source — unlike F5′s, Kong's, and HAProxy's Ingress-only answers, that CRD field is reachable from *both* the classic path and `HTTPRoute` (via the same generic `ExtensionRef` bridge covered below). So this bifurcation isn't unique to NGINX: any target that answers `rewrite-target` via its own CRD/annotation on its own data plane tends to carry regex over, while every target that speaks only core Gateway API `URLRewrite` does not.

The classification scheme

Each row in the matrix gets an overall, annotation-level **risk** (Green / Yellow / Red) in the CSV, describing how dangerous or complex *the annotation itself* is to migrate, independent of any specific target — plus an `unverified` state for cases that couldn't be confirmed from a primary source, rather than a guess dressed up as an answer. A full re-verification pass against every target's own source resolved every cell that still carried that label from earlier drafts of this matrix — none of the 17 rows below need it right now — but the state stays in the scheme as the matrix grows past 17.

The grid below is a different, finer-grained view: not the annotation-level risk above, but a per-target verdict — for each of the eight targets, does *that specific controller* carry this annotation's behaviour over cleanly? Green is a clean, low-friction equivalent; yellow is a real path with real friction (a CRD or plugin instead of a plain annotation, partial support, a licensing gate); red is a confirmed absence, rejection, or silent no-op; gray is reserved for unverified or structurally-not-applicable cells — none appear in the grid below right now. Every cell traces back to a specific finding in the sections and the CSV below — this is a summary, not a separate research pass. The last row is a different question and deliberately breaks the color pattern: not "does this migrate

cleanly" but "which process actually proxies the traffic" — there's no good/bad answer to that one, just an identity, so it uses its own categorical colors instead of the risk palette above it.

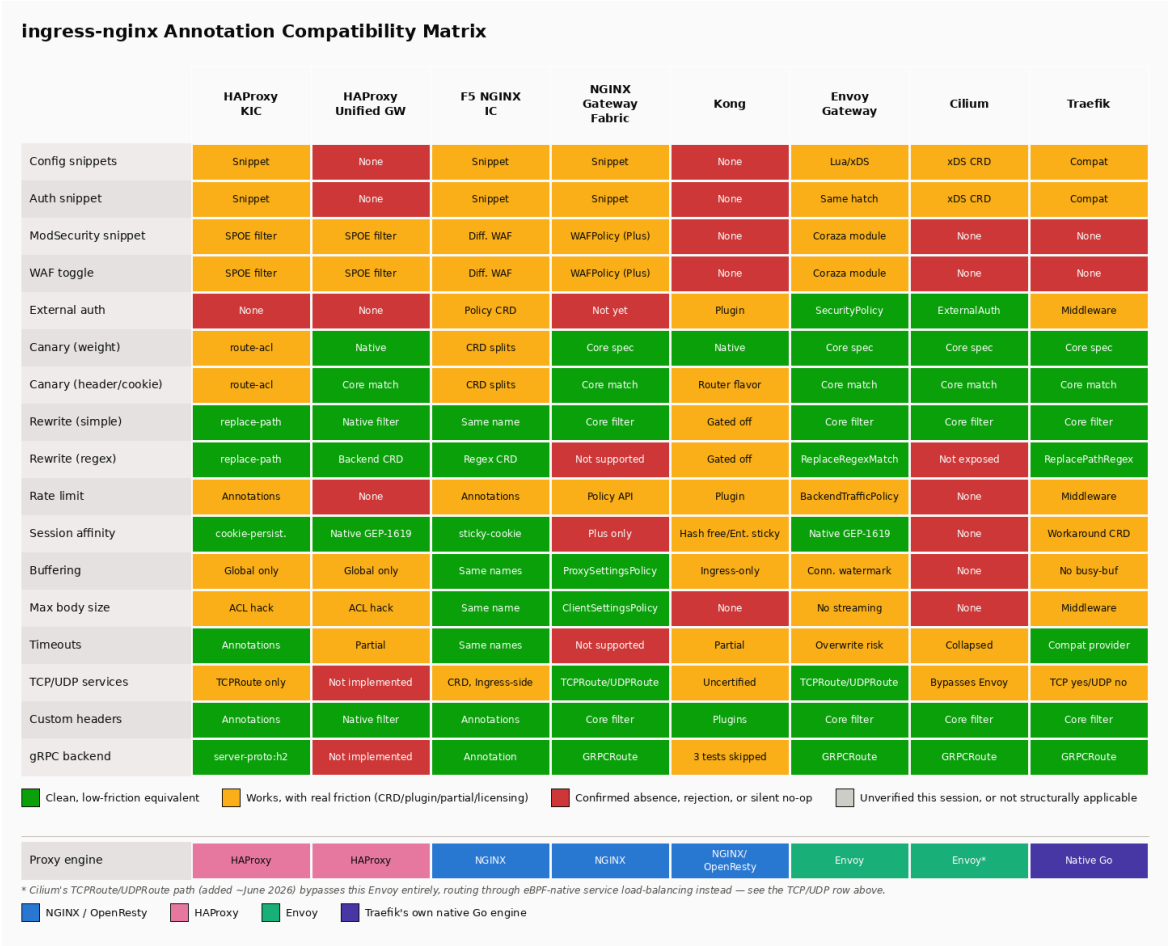

Figure 1: 17 hard-case annotations x 8 migration targets, color-coded green/yellow/red/gray, plus a proxy-engine identity row

Two migration paths, eight targets

The matrix now evaluates eight targets, and they don't all represent the same kind of move. Part 1 named two separate paths — "switch to another Ingress controller" and "move to Gateway API" — and most targets split cleanly across that line:

- **Switch to another Ingress controller** (same object model; existing `Ingress` resources mostly keep working, usually with annotation-name changes): the HAProxy Kubernetes Ingress Controller, and F5's own NGINX Ingress Controller (`nginx/kubernetes-ingress`). That name needs unpacking: it is neither the retired `ingress-nginx` this whole series is about, nor `nginx-gateway-fabric` two paragraphs from here — three separate NGINX-adjacent projects, easy to conflate, worth being precise about.
- **Move to Gateway API** (a different object model — `HTTPRoute`/`GRPCRoute`/etc. replace `Ingress` entirely): Envoy Gateway, NGINX Gateway Fabric, and HAProxy's own **HAProxy Unified Gateway** (`haproxytech/haproxy-unified-gateway`, "HUG"). Confirmed directly from Envoy Gateway's own source: it contains no reference anywhere to the `networking.k8s.io/v1`

Ingress types, and its own README describes it as managing "Envoy Proxy as a standalone or Kubernetes-based application gateway" — not as an Ingress controller. Evaluating it in this matrix is correct precisely *because* it isn't one: it's the reference implementation for the Gateway API path Part 1 already named as a legitimate option on its own. HUG is the same story from HAProxy's side: a genuinely separate project from the `haproxy` column above it, built against Gateway API only (its own README says Ingress-API support is "coming next year"), confirmed from source.

Three targets don't split cleanly at all, though: Cilium, Traefik, and **Kong's Kubernetes Ingress Controller** (`kong/kubernetes-ingress-controller`) support the classic Ingress API and Gateway API side by side — either, depending on configuration. They get there by three different architectures, all confirmed from source. Kong is the most tightly integrated: one single codebase and binary runs the Ingress reconcilers (`internal/controllers/configuration/`) and the Gateway API reconcilers (`internal/controllers/gateway/`) next to each other, independently toggle-able by CLI flag. Cilium's Ingress and Gateway API ingestion code are genuinely separate entry points, but they converge into one shared `model.Model` before either ever reaches Envoy — everything from that shared model onward (cluster/listener/route-configuration generation) is one code path, not two paths that merely produce compatible output. Traefik keeps the two furthest apart: its classic provider (`pkg/provider/kubernetes/crd/`) and its Gateway API provider (`pkg/provider/kubernetes/gateway/`) are separate packages with separate reconcilers, bridged only by a generic `ExtensionRef`→`kind: Middleware` registry — every Middleware capability (auth, rate limiting, buffering, regex rewrite) reaches `HTTPRoute` through that one bridge, not per-feature glue code.

Kong earns its spot in this matrix for a different reason than F5's or HAProxy's additions below: it's one of the officially supported providers listed in `kubernetes-sigs/ingress2gateway` — the same upstream signal this series already treats as a legitimacy bar for a target being worth evaluating at all, confirmed by cloning that repo directly (`pkg/i2gw/providers/kong/`) rather than trusting its README's provider list at face value.

Two vendors now show up on both sides of that Ingress-vs-Gateway-API line at once, in two entirely separate codebases: NGINX has an Ingress controller *and* a Gateway API implementation, and so does HAProxy. Kong's shape is different again — one codebase answering both APIs. Worth checking not just which column a vendor's name lands in, but which of these three shapes it takes, before assuming what "migrating to X" actually involves.

There's a fourth path this matrix hadn't accounted for either, and it belongs to Traefik alone: **consume the existing `nginx.ingress.kubernetes.io/*`-annotated `Ingress` objects directly, without switching object model at all.** Confirmed from source (`pkg/provider/kubernetes/ingress-nginx/`): Traefik ships a dedicated compatibility provider that parses ingress-nginx's own annotations — including, for `server-snippet`/`configuration-snippet`, a real nginx-config parser that re-executes a bounded allow-list of directives (`set`, `rewrite`, `proxy_set_header`, `add_header`, and about a dozen others; anything outside that list is a hard build-time failure, not a silent pass-through). No other target researched in this series has anything comparable — every other target's answer to "what happens to my existing `Ingress` objects" is "translate them," not

"run them as-is." Worth treating as its own migration path in a future part, not folding into "switch Ingress controller" — the whole point of that path elsewhere in this series is annotation-name changes; this path is annotation compatibility.

This matters because the migration effort is not the same shape across the two paths — switching Ingress controllers can often keep your existing `Ingress` objects with just annotation-name changes; moving to Gateway API means re-modeling routing as new object types from scratch. The per-row findings below hold regardless of category, but worth keeping in mind when a cell says "supported."

F5′s own NGINX Ingress Controller earns a specific callout on the fairness point: it runs the same underlying NGINX data plane as the retired `ingress-nginx`, and several of its annotations are close to identical — `nginx.org/rewrite-target`, `nginx.org/proxy-read-timeout`, `nginx.org/client-max-body-size`, just under a different prefix than `nginx.ingress.kubernetes.io/`. Of every target researched in this series so far, it's the one most likely to be the lowest-effort migration for a given annotation — not because it's "the best" target in the abstract, but because it shares the most heritage with what you're migrating away from.

HUG earns the same kind of callout for the opposite reason: it's the *least*-effort answer to recreate, not the least effort to *reach*. It's a young project — its own conformance testing declares only `SupportGateway`, `SupportHTTPRoute`, and `SupportReferenceGrant`, and this session's source review found several core-spec gaps consistent with that (no GRPCRoute/TCPRoute/UDPRoute reconciliation yet, a partially-implemented `Timeouts` field, no rate-limit or external-auth mechanism at all). What it does implement, it implements natively against the Gateway API spec itself rather than through a custom Policy CRD layered on top — most visible in session affinity, below. Evaluating it here isn't an endorsement of maturity; it's the same fairness argument as F5′s column: if one vendor's Gateway API implementation is in the matrix, the other vendor's should be too, immature or not.

Kong's standout finding is architectural rather than a single feature: across nearly every hard case below — rate limiting, external auth, WAF, canary via a dedicated plugin — Kong's own Kubernetes Ingress Controller has **no dedicated code path of its own** for the feature. Instead, almost everything funnels through one generic bridge, confirmed from source: the `konghq.com/plugins` annotation on `Ingress`, and the equivalent `ExtensionRef` filter pointing at a `KongPlugin` CRD on `HTTPRoute`, both attaching the same underlying plugin object regardless of which API you're routing through. That's a genuinely different shape from every other target in this matrix, where "does X exist" usually means "is there a dedicated CRD/annotation/filter for X." For Kong, the more useful question is almost always "does a Kong Gateway plugin exist for X" — and that answer lives outside this controller's own repository, in Kong Gateway itself.

The following tables compare a selection of ingress-nginx features and make no claim to completeness.

Snippet annotations: still the clearest “no”

`configuration-snippet`, `server-snippet`, `stream-snippet`, and `auth-snippet` inject raw NGINX configuration directly. Every one of them is rated **Critical** in ingress-nginx’s own annotation risk table — this isn’t just a migration problem, it’s already the project’s own top operational risk category. None of the eight targets researched here has a *portable* equivalent, by design:

> ☒ **Required framing for every snippet annotation**
>
> No portable equivalent by design. Treat every snippet as custom proxy code, not as configuration.

What each target actually offers instead is informative, though, because none of it is “no equivalent, full stop” — it’s “no *portable* equivalent, but here’s the escape hatch, and here’s what it costs”:

Target	Escape hatch	Confirmed from
Envoy Gateway	`EnvoyExtensionPolicy` (Lua/Wasm), `EnvoyPatchPolicy` (raw xDS patches)	2026 RCE CVE in the Lua path (GHSA-xrwg-mqj6-6m22, CVE-2026-22771)
Cilium	`CiliumEnvoyConfig` (raw Envoy xDS)	Explicitly unvalidated by the Kubernetes API — source marks the resource body `x-kubernetes-preserve-unknown-fields`; a malformed config is admitted and only fails later, in agent logs
NGINX Gateway Fabric	`SnippetsFilter`	Same NGINX data plane as ingress-nginx; framed as "still custom proxy code"; open bugs (NGF #4616)
F5 NGINX Ingress Controller	`server-snippets`/ `location-snippets`	Closest lineage of all eight targets — same NGINX core, same risk category
HAProxy Kubernetes Ingress Controller	`*-config-snippet` family + a full secondary `haproxy.cfg` via ConfigMap	`documentation/ secondary-config.md`
HAProxy Unified Gateway	**None**	`Global`/`Defaults`/`Backend` CRDs are fully structured fields only — inverse trade-off from its own vendor's Ingress-API controller above
Kong Kubernetes Ingress Controller	**None** dedicated	`KongCustomEntity` is schema-typed, not freeform; the real risk-equivalent is Kong Gateway's own `pre-function`/`post-function` plugins, reached via the generic plugin bridge, not KIC code
Traefik	**None** on the native path — but see below	Dedicated ingress-nginx-compat provider, unique among all eight targets

The Lua RCE and the raw-xDS escape hatches carry real risk, same class as nginx snippets. HAProxy Unified Gateway is the one structural inverse: no snippet mechanism at all, at the cost of no way to reach outside the schema either. Traefik is the standout, though: it ships a dedicated ingress-nginx-annotation compatibility provider (`pkg/provider/kubernetes/ingress-nginx/`, confirmed from source) that directly parses `server-snippet`/`configuration-snippet` text with a real nginx-config parser — genuinely bounded, not a like-for-like carryover: any directive outside a fixed allow-list of about 16 (`set`, `rewrite`, `proxy_set_header`, `add_header`, and a handful more) fails the build outright rather than silently passing through. Separately, Traefik's own Yaegi plugin system does interpret real Go code — including `syscall`/`unsafe` — in-process, a genuine arbitrary-code-execution surface, but plugins are operator-installed via static config, not something a tenant can inject through an `Ingress` annotation the way an nginx snippet is.

Worth noting separately: Traefik runs its own Go-native proxy core, not NGINX — so even the parsed `server-snippet`/`configuration-snippet` text on the compat path is being translated into Traefik's own configuration model, not passed through to an actual nginx.conf.

`modsecurity-snippet` gets its own row in the matrix for the same reason — it's Critical risk on ingress-nginx too, and distinct from the WAF *toggle* itself (`enable-modsecurity`, `enable-owasp-core-rules`, Low risk upstream), which is a separate, if still Yellow-rated, migration question: WAF is handled by every target through its own extension mechanism, none of them carry ModSecurity rule syntax over directly.

auth-url / auth-signin: no stable answer, several mature-ish ones

External auth delegation has no stable core Gateway API filter — an `ExternalAuthFilter` exists, but only in the experimental channel, and it's explicitly subject to change. Confirmed directly from the Gateway API source: the core `HTTPRouteFilterType` enum does **not** include `ExternalAuth`; it only appears in the experimental validation enum. What each target does instead:

Target	Mechanism	Note
Envoy Gateway	`SecurityPolicy` (ExtAuth)→ Envoy `ext_authz`	Most battle-tested — Envoy Gateway is the reference implementation team for `ext_authz` itself
Cilium	Core (experimental) `ExternalAuthHTTPRoute` filter → the same `ext_authz` filter	Most standards-native answer in this row — the only target reaching auth through the actual spec field, not a vendor Policy CRD; Gateway API side only, no `Ingress`-side equivalent
F5 NGINX Ingress Controller	`Policy` CRD `externalAuth` block(`authSigninURI`, `authSigninRedirectBasePath`)	Real, mature, documented — via CRD, not a plain annotation
Traefik	Compat provider auto-translates to `dynamic.Auth`(narrower type, `pkg/provider/kubernetes/ingress-nginx/middleware.go`)	The generic `Middleware` CRD's own richer `ForwardAuth` field (TLS, `ForwardBody`, `MaxBodySize`) is real and reachable via `ExtensionRef`→`kind: Middleware` — but that's a separate, hand-configured capability, not what these annotations auto-wire to
NGINX Gateway Fabric	None native yet	Open issue (#4485); building a bespoke `AuthenticationFilter`
HAProxy Kubernetes Ingress Controller	**None**`auth-type` supports only `basic-auth`	Corrected finding: `oauth-*` annotations found via search belong to the *unrelated* `haproxy-ingress/haproxy-ingress` project, not this one — see note below
HAProxy Unified Gateway	**None**filters limited to `RequestHeaderModifier`/`ResponseHeaderModifier`/`URLRewrite`/`RequestRedirect`/`CORS`	Experimental `ExternalAuth` filter unimplemented; `ExtensionRef` only resolves to its own `Backend` CRD
Kong Kubernetes Ingress Controller	**None** dedicatedan auth plugin via the generic `konghq.com/plugins`/`ExtensionRef` bridge	Actual auth logic lives in Kong Gateway, outside this repository

Two findings worth the extra sentence: Cilium's is the *only* answer in this row running through the real (if still experimental) Gateway API spec field rather than a vendor-specific Policy CRD/Middleware — every other target, including Envoy Gateway's own, layers its own CRD on top. And the HAProxy row is a corrected finding, not a first pass: an earlier version of this research attributed `oauth`/`oauth-uri-prefix`/`oauth-headers` annotations to `haproxytech/kubernetes-ingress` — those belong to a *different*, confusingly near-identically-named project

(`haproxy-ingress/haproxy-ingress`). Conflating the two is an easy mistake from search results alone; cloning the actual repo is what caught it.

Canary: a genuine split inside one hard case

Weight-based canary (`canary`, `canary-weight`) maps directly onto Gateway API's core weighted `backendRefs` — stable since v0.5.0, one of the cleanest rows in this matrix. Header/cookie-based canary (`canary-by-header`, `canary-by-cookie`) is a different mechanism entirely: there's no weight annotation to port, because the replacement is architectural — two separate `HTTPRoute` objects with header matches pointing at different backends, not one object with a routing-weight annotation. Cookie matching has no native match type in core Gateway API at all (`HTTPRouteMatch` supports only `Path`, `Headers`, `QueryParams`, and `Method`), which is why most Gateway-API-native targets in the table below show "no cookie match." Two targets get there anyway, each a different way: HAProxy and F5 sidestep Gateway API entirely and use their own ACL/CRD mechanism instead (see below), while Envoy Gateway stays on Gateway API but bolts cookie matching on as its own vendor extension. Same feature category, same prompt-doc "hard case" label, genuinely different migration shape.

Target	Weight-based	Header/cookie-based
HAProxy Kubernetes Ingress Controller	Single `route-acl` annotation, in-line ACL expression (e.g. `rand(100) lt 25`)	Same `route-acl` annotation, different expression (e.g. `cookie(staging) -m found`) — one mechanism for both
F5 NGINX Ingress Controller	`VirtualServerRoute` CRD `splits[]`	Same CRD, combined with `matches[]` — one richer mechanism instead of several annotations
HAProxy Unified Gateway	Native `backendRefs` weights, end-to-end (`k8s/gate/haproxy/routes.go`) — clean win	Core `HTTPRoute` match only, no cookie match, no extra ergonomics
Kong Kubernetes Ingress Controller	Native `backendRefs` → Upstream `Target` weights (`translate_upstreams.go`) — clean win, no dedicated canary plugin	Core matches work, but header matching is skipped under the older "traditional_compatible" router flavor (issue #6144); no cookie match — closest path is `KongUpstreamPolicy` (see session affinity)
Envoy Gateway	Native `backendRefs` → Envoy weighted clusters	Header matching is core Gateway API; cookie matching is **not** core but works anyway — Envoy Gateway ships its own vendor extension (`HTTPRouteFilter.spec.match.cookies`) — the only target in this matrix with a working, if non-portable, cookie match
Cilium	Native `backendRefs` → Envoy `WeightedCluster` for `HTTPRoute`/`GRPCRoute`; for `TCPRoute`/`UDPRoute`, weighting still works but through a different mechanism — Cilium's own eBPF Maglev load-balancer instead of Envoy (see TCP/UDP section)	Core matches only, no cookie match, no canary-specific ergonomics
Traefik	Native weighted round-robin (`loadWRRService`), conformance-passing in full	Core matches only, no cookie match, no router-flavor split the way Kong has
NGINX Gateway Fabric	Native `backendRefs`, standard Gateway API conformance	Core header matches, standard conformance — no cookie match, same core-spec limitation as every other target here

HAProxy and F5 both collapse weight- and match-based canary into one richer mechanism (an ACL expression, a CRD field) rather than several dedicated annotations — neither is more "portable" than the other, just shaped differently from ingress-nginx's own several-annotation approach.

Timeouts: where the matrix earns its keep

This is the sharpest finding in the batch. Gateway API's core `Timeouts` field (`request` and `backendRequest`, GA since v1.2.0) looks like a clean, portable answer to `proxy-connect-timeout`/`proxy-read-timeout`/`proxy-send-timeout`. Whether it actually *works* depends entirely on the target:

Target	backendRequest	request	Verdict
Cilium	✅	✅	Both fields are read from the spec, but collapsed via `min(backendRequest, request)` before rendering — into Envoy's single `RouteAction.Timeout` — no `RetryPolicy.PerTryTimeout` wired at all. Not a clean pass: if `backendRequest` < `request` (the spec-intended case), the shorter value silently becomes the *overall* timeout, and retries get no separate per-try budget
NGINX Gateway Fabric	❌	❌	Rejects outright — own docs say "Not supported." Visible failure.
Traefik	❌ (core path) / ✅ (compat provider)	❌ (core path) / ✅ (compat provider)	Two different answers depending on migration path: hand-authoring core Gateway API `Timeouts` is silently ignored — confirmed from Traefik's own checked-in conformance report (`unsupportedFeatures`), `HTTPRouteRule.Timeouts` never read in the provider at all. But Traefik's own ingress-nginx-compat provider — the realistic migration path for *these specific annotations* — maps `proxy-connect/read/send-timeout` 1:1 onto `ServersTransport.ForwardingTimeouts`, a working per-Ingress override
Envoy Gateway	✅	✅	Reads both fields, but `BackendRequest` unconditionally overwrites `Request` unless a `Retry` block is also configured — last-write-wins, not `min()`. Only with `Retry` configured does `BackendRequest` correctly become the per-retry timeout (GEP-1742) while `Request` stays the overall one. Without `Retry`, setting both silently drops `BackendRequest`. Ingress-side, `proxy-connect-timeout` (outside this core field entirely) maps
[illegible]	n/a (maps via [illegible]	n/a	[illegible]

The sharpest read has to be revised from an earlier pass on this same row. NGF's rejection is still a *visible* failure, and HUG/Kong still split the difference — half the field silently does nothing rather than all of it. But Traefik's core-Gateway-API silent no-op is no longer the row's cautionary tale once the compat provider is on the table: for the actual annotations this row is about, Traefik's real migration path works fine. Cilium and Envoy Gateway aren't clean exceptions either — both read both fields, but neither composes them the way the spec intends. Cilium collapses them into one Envoy timeout via `min()` before either reaches the proxy; Envoy Gateway lets one field unconditionally overwrite the other unless a `Retry` block happens to be present too. No target in this matrix gives both sub-fields independent, always-honored semantics. The row's real lesson: "the field exists and gets read" is not the same claim as "the field's semantics survive translation" — check the translation step, not just the schema.

WebSocket: hiding inside the timeout row

WebSocket gets used daily and understood rarely, which is exactly why it earns its own callout here instead of a passing mention. There is no dedicated WebSocket annotation in ingress-nginx — `Upgrade`/`Connection` headers pass through by default — so WebSocket doesn't show up as its own row anywhere in this matrix. It hides inside the timeout group above: a WebSocket connection is a single long-lived HTTP request, and whether it survives depends entirely on whether a target enforces a long-enough value for the specific timeout field that governs it. Checking "does this target support WebSocket" is the wrong question; checking "does this target enforce the timeout value I actually set" is closer, but the table above shows even that isn't precise enough.

Two targets turn this into a silent trap rather than a simple pass/fail. Cilium's `min(backendRequest, request)` collapse means whichever of the two fields is shorter wins — set a long `request` for your WebSocket route and leave `backendRequest` at a short default, and the short value silently caps the connection, with no error anywhere. Envoy Gateway has the same practical effect through a different mechanism: set both fields without also configuring a `Retry` block, and `backendRequest` unconditionally overwrites `request` — the long value you tuned for WebSocket is the one that gets discarded. HAProxy Unified Gateway and Kong are more honest about the same underlying gap: `request` isn't implemented at all, so there's no ambiguity about which field to tune, only `backendRequest`. NGF's outright rejection of the whole field is, perversely, the safest failure mode of the five — it fails loudly instead of quietly picking the wrong number.

Session affinity: a licensing gate hiding behind a feature-support answer

`affinity: cookie` has no GA core Gateway API equivalent — `BackendLBPolicy` session persistence (GEP-1619) is experimental-channel only. Looking at implementation support:

Target	Mechanism	Free tier?
Envoy Gateway	Native experimental `SessionPersistence` field (GEP-1619) — cookie-based (default) or header-based, with `AbsoluteTimeout` support. A separate `BackendTrafficPolicy` consistent-hashing mechanism also exists, but that's deterministic routing, not true session persistence, and isn't what answers this row	Yes
Cilium	**None** — spec field never read (zero grep hits), no alternative CRD either	n/a — only a hand-assembled `CiliumEnvoyConfig` using Envoy's own `ring_hash`/`maglev` policy
NGINX Gateway Fabric	Session persistence supported	**NGINX Plus only** — no OSS equivalent
Traefik	Spec field not implemented, absent even from its own conformance report — but a real workaround is reachable from `HTTPRoute`: `backendRef` → `TraefikService` CRD, `Weighted.Sticky`	Yes, via the workaround
HAProxy Kubernetes Ingress Controller	`cookie-persistence`/`cookie-persistence-no-dynamic`	Yes, no caveat
F5 NGINX Ingress Controller	`sticky-cookie-services` (`nginx.org/`-prefixed)	Yes — free tier isn't crippled the way NGF's is
HAProxy Unified Gateway	Native experimental `SessionPersistence` field (GEP-1619) — cookie name, absolute/idle timeout	Yes — one of two targets in this matrix (with Envoy Gateway) reaching session affinity through the actual spec field itself, not a Policy CRD
Kong Kubernetes Ingress Controller	`KongUpstreamPolicy`: consistent-hashing (`hash_on: cookie`, etc.)	Hashing free; true cookie-based sticky-sessions **Enterprise only**

Three things worth the extra sentence: NGF's Plus-only gate is a product decision by the Gateway Fabric team, not something inherent to NGINX as a data plane or to Gateway API as a spec — F5′s own classic Ingress controller proves the free tier can do this fine. Kong sits in a more granular version of the same trap: "does Kong support session affinity" is yes for the hashing flavor and no for the classic sticky-cookie flavor unless you're on the paid tier. And HAProxy Unified Gateway and Envoy Gateway both implement the real GEP-1619 field natively — two different vendors, same observation: everyone else answers this row through their own custom Policy CRD, not the spec field Gateway API actually defines for it. A straight annotation-name lookup would miss all three nuances.

Canary vs. session affinity: two different “cookie” fields

Both this row and the `canary-by-cookie` case above use the word “cookie,” which invites treating them as the same underlying capability — they aren’t. Canary’s cookie-based *routing* has no core match type at all: `HTTPRouteMatch` only defines `Path`, `Headers`, `QueryParams`, and `Method`, so reaching it means going through a vendor extension, like Envoy Gateway’s own `HTTPRouteFilter.spec.match.cookies`. Session affinity’s cookie is a different field entirely, `HTTPRouteRule.SessionPersistence` (GEP-1619) — it lives in the core spec (experimental channel) and doesn’t route on the cookie’s value at all; it just tells the proxy to *set* one so a client’s next request lands back on the same backend. Same word, two unrelated fields, two unrelated levels of core-spec support — Envoy Gateway, for instance, needs its own extension for the first and implements the second natively.

TCP/UDP services: this changed under the series’ feet

Gateway API’s `TCPRoute`/`UDPRoute` graduated to the Standard (GA) channel in **v1.6.0** (2026-06-29) — confirmed directly from the Gateway API repository, where `TCPRoute` now lives in the stable `apis/v1` package rather than `v1alpha2`. That’s newer than most existing Ingress-vs-Gateway-API comparisons account for. ingress-nginx’s `tcp-services`/`udp-services` ConfigMaps (Part 2′s Surface 2, not a per-Ingress annotation) are the source-side feature all of this maps to.

Target	TCPRoute	UDPRoute	Note
Envoy Gateway	✅	✅	Confirmed, v1.7.0
NGINX Gateway Fabric	✅	✅	Per NGF’s own compatibility docs
Cilium	✅ (landed ~June 2026)	✅ (same)	**Bypasses the embedded Envoy entirely** — see below
Traefik	✅ (alpha API group, full reconciler)	✅	Classic `IngressRouteUDP` CRD still covers UDP, just disconnected from the Gateway API kind
HAProxy Kubernetes Ingress Controller	Claimed, per (stale) docs	Unclear	Docs dated 2023, code at v3.2.12 — unverified-current, see below
F5 NGINX Ingress Controller	n/a — Ingress-side `TransportServer` CRD instead		Richer, per-resource, with health checking; not a Gateway API claim
HAProxy Unified Gateway	❌	❌	Route-kind constants exist, no reconciler behind either
Kong Kubernetes Ingress Controller	❌ (not in the certified conformance profile)	❌ (same)	Legacy `TCPIngress`/`UDPIngress` CRDs also still default-enabled, despite being marked deprecated

Three findings need the extra sentence. **Cilium only recently started reconciling `TCPRoute`/`UDPRoute` at all** — support landed in Cilium's own codebase around June 2026, after a long stretch where this was tracked only as an open CFP issue — and the new path deliberately routes around Envoy, materializing a plain `LoadBalancer` Service that Cilium's eBPF service load-balancer consumes directly. Traffic weighting still works on this path — confirmed from source (`l4HasWeights`, `operator/pkg/model/translation/gateway-api/ translator.go`): whenever a backend carries a weight, Cilium switches the Service's load-balancing algorithm to Maglev, which is required for its datapath to honor weights at all. The real caveat is elsewhere: only the single oldest-by-creation-timestamp route per listener is honored, everything else attached to that listener is silently dropped (see the canary section above for the same mechanism applied to weighting). The generalizable lesson: even a source-grepped answer can go stale between one research pass on a fast-moving project and the next — re-verify close to publication. **HAProxy's own docs are stale, and the reason why is bigger than a docs bug**: `gateway-api.md` states TCPRoute-only support on Gateway API v0.5.1, and the one and only commit that ever touched that file is dated 2023-02-23. Since then, this controller's own Gateway API code has only received maintenance-level commits — an install-detection fix, an error-message cleanup, RBAC handling — no feature growth. That timeline lines up with HAProxy's real Gateway API investment moving into a separate, purpose-built project instead: HAProxy Unified Gateway, already covered elsewhere in this matrix, whose own README describes it as the "latest Kubernetes Gateway API" answer for HAProxy. Read together, this isn't an abandoned feature so much as HAProxy having concluded that Gateway API needed its own implementation rather than a bolt-on to the existing Ingress controller — either way, a project's own docs page can be stale enough to actively mislead; check the commit history, not just the page. **Kong answers this on both APIs at once**, a small case study in its dual-codebase architecture — functional on the Gateway API side, just not yet part of the conformance profile its own test suite exercises.

QUIC and HTTP/3: not in this matrix, and here's why

Unlike WebSocket, HTTP/3 doesn't hide inside another row here — it's absent because there's genuinely nothing on the source side to migrate yet. ingress-nginx ships an nginx build with the `--with-http_v3_module` flag compiled in, but that's step one of five on the project's own roadmap (`images/nginx/README.md`): no ConfigMap flag, no `listen ... quic` directive in the server template, no opened UDP port, explicitly labelled "experimental and under development," confirmed by zero HTTP3-related hits anywhere in `internal/ingress/controller/config/config.go`. No annotation, no ConfigMap key, nothing a migration could carry over — so this matrix has no row for it, the same way it has no row for any other feature that doesn't exist.

Worth a forward-looking mention anyway, since a controller migration is also a chance to pick up capabilities ingress-nginx never had. Two targets researched in this series already support HTTP/3 natively: Envoy Gateway exposes it as a first-class `HTTP3` field on `ClientTrafficPolicy`, and Traefik ships its own dedicated HTTP/3 entrypoint (`pkg/server/server_entrypoint_tcp_http3.go`) as a long-standing feature. Cilium's Gateway API/Ingress path showed no HTTP/3 wiring anywhere in the same source areas checked for every other row in this matrix. None of this changes the row-by-row comparisons above — HTTP/3 isn't a

compatibility question when the source never had the feature — but it's a real difference between targets if HTTP/3 is on the roadmap independently of this migration.

Two clean wins, so this doesn't read as uniformly bad news

Not everything here is yellow or red. `backend-protocol: GRPC` maps directly onto Gateway API's `GRPCRoute` — its own dedicated, GA, core route type since v1.1.0 — arguably the single cleanest migration in this entire matrix. Custom response headers are a genuine upgrade too: ingress-nginx's `custom-headers` annotation requires a referenced ConfigMap; Gateway API's core `RequestHeaderModifier`/`ResponseHeaderModifier` filters (a Standard-channel conformance requirement) do it inline, no indirection needed.

Target	`GRPCRoute`	Ingress-side gRPC equivalent	Header modifiers
Envoy Gateway	✅	n/a	✅ clean
NGINX Gateway Fabric	✅	n/a	✅ clean
Cilium	✅ — shares the `model.HTTPRoute` translation pipeline wholesale, not a parallel gRPC-only path	n/a	✅ clean
Traefik	✅ conformance-passing (`GATEWAY-GRPC` profile, all core tests)	n/a	✅ clean
HAProxy Kubernetes Ingress Controller	n/a — its own Gateway API support is TCPRoute-only (see TCP/UDP section)	`server-proto: h2` sets clear-text HTTP/2 to the backend — exactly what gRPC needs	✅ clean (`request-set-header`/`response-set-header`)
F5 NGINX Ingress Controller	n/a	`grpc-services` (also has an explicit `websocket-services` annotation, unlike ingress-nginx)	`proxy-set-headers`/`add-header`
HAProxy Unified Gateway	✅ — same fate as `TCPRoute`/`UDPRoute`, a route-kind constant with no reconciler	n/a	✅ clean (`k8s/gate/haproxy/filters/http_filters.go`)
Kong Kubernetes Ingress Controller	✅ core-conformant (3 tests skipped, issue #7919)	`konghq.com/protocols`	✅ clean (`request-transformer`/`response-transformer` plugins)

Header modifiers are about as close to a universally-solved row as this matrix has — every target researched implements them cleanly, with no partial or missing cases. gRPC is just as clean on the Ingress side — every target has a real, working answer — and the only genuine gap is HAProxy Unified Gateway, for the same structural reason as its `TCPRoute`/`UDPRoute` absence above.

Companion repo

The full 17-row CSV — now against **eight** targets, including rate limiting, buffering, body size, and the two clean-win rows covered only briefly above — lives in the `ingress-nginx-migration-lab` companion repo at `matrix/ingress-nginx-annotation-migration.csv`, generated from `scripts/build-annotation-matrix.py` rather than hand-edited (safer for a 20-column CSV). The heatmap image at the top of this part is generated the same way, from `scripts/generate-matrix-heatmap.py` — a visual summary of that CSV, not a separate source of truth. This is a first batch, not the full 100+ annotations ingress-nginx ships — it's a living document that grows with later parts of this series, and the unlisted annotations aren't a secret, just not yet researched to the same bar.

What's next

Part 4 picks candidates based on what this matrix actually found, not on feature-list marketing — which also means Part 4 doesn't start until the categories used here are confirmed. The series hub tracks what's published; Part 2 covers building the inventory this matrix classifies.

None of this matrix decides anything by itself, though. It tells you where an annotation migrates cleanly and where it doesn't — the actual call between staying on ingress-nginx via a maintained fork, switching to another Ingress controller, or moving to Gateway API depends on criteria this data can't supply: which of these rows your own `Ingress` objects actually depend on, how much custom-proxy-code risk is acceptable, and how the migration effort weighs against everything else competing for the same engineering time. Finding that weighting is on the reader, not on this matrix.

Sources

- ingress-nginx annotations-risk.md — kubernetes/ingress-nginx (main branch)
- kubernetes/ingress-nginx — ingress-nginx (source, main branch; confirms the absence of a working HTTP/3/QUIC feature beyond a compiled-in nginx module)
- Gateway API HTTPRoute reference — gateway-api.sigs.k8s.io
- kubernetes-sigs/gateway-api — Gateway API (source, main branch; confirms core-vs-experimental field status, API version history, and Standard/Experimental channel graduation dates directly from `apis/v1` and `versioning.md`)
- HTTP path redirects and rewrites — gateway-api.sigs.k8s.io
- Gateway API v1.6.0 release — kubernetes-sigs/gateway-api (2026-06-29)
- Envoy Gateway SecurityPolicy — gateway.envoyproxy.io
- Envoy Gateway BackendTrafficPolicy — gateway.envoyproxy.io
- EnvoyExtensionPolicy Lua RCE advisory (GHSA-xrwg-mqj6-6m22) — envoyproxy/gateway
- NGINX Gateway Fabric Gateway API compatibility — docs.nginx.com
- Traefik Kubernetes Gateway API reference — doc.traefik.io
- Traefik issue #11902 — GEP-1742 Timeouts — traefik/traefik
- Traefik issue #11243 — sticky sessions via Gateway API — traefik/traefik
- Cilium Gateway API support — docs.cilium.io

- cilium/cilium — Cilium (source, main branch; confirms the embedded/per-node Envoy architecture, the shared Ingress/Gateway API translation model, and the TCPRoute/UDPRoute reconciler added ~June 2026)
- traefik/traefik — Traefik (source, main branch @ v3.7.8+; confirms the native Go reverse-proxy engine, the generic `ExtensionRef`→`Middleware` bridge, the dedicated ingress-nginx-annotation-compatibility provider, and includes Traefik's own checked-in Gateway API conformance report)
- haproxytech/kubernetes-ingress — HAProxy Kubernetes Ingress Controller (source, v3.2.12)
- envoyproxy/gateway — Envoy Gateway (source, confirms no Ingress API support)
- nginx/kubernetes-ingress — F5 NGINX Ingress Controller (source, v5.5.3)
- haproxytech/haproxy-unified-gateway — HAProxy Unified Gateway "HUG" (source, v1.0.5)
- kong/kubernetes-ingress-controller — Kong Kubernetes Ingress Controller (source, main branch)
- kubernetes-sigs/ingress2gateway — upstream provider list that surfaced Kong as a target worth evaluating (source, v1.2.0)