

Inbound Traffic Policy

Shared with Istio Community

	
Owner: Liam White Working Group: Networking	Status: WIP In Review Approved Obsolete Created: 2025-08-26 Approvers:

Overview

Goals

- Provide inbound/server-side listener configuration without the [EnvoyFilter](#) API, for better ergonomics and guarantees during upgrades.
- Support the relevant configuration uniformly across all Istio data planes: Istio Gateway, sidecars, waypoints, and Kube Gateway API.
- Follow the GEP-713 policy attachment model for consistency with existing Istio APIs such as [Telemetry](#) and [TrafficExtension](#).

Non-Goals

- Ship with every field populated at once: the resource ships with [http.compression](#) only in its initial release. Connection configuration, local rate limiting, and global rate limiting land as fast-follows, each with its own design doc (see other tabs)
- Configuring hop 2: the proxy's own connection to its local app or backend pods ([Envoy → app](#)). That remains [DestinationRule](#)'s domain. See [Scope: Hop 1 vs. Hop 2](#) for the full reasoning.

Background

Several categories of inbound/server-side proxy configuration have no native declarative Istio API today: HTTP response compression, connection-level admission/lifecycle limits, and rate limiting all currently require the [EnvoyFilter](#) API, writing raw Envoy configuration directly.

The [EnvoyFilter](#) API is considered the API of last resort and should be avoided where possible because using it involves breaking through the Istio abstraction and interacting with Envoy APIs directly. This makes it difficult for Istio to provide any guarantees around backwards compatibility: upgrades can cause functionality to break due to Envoy API version changes, filter name changes, filter ordering changes, and so on.

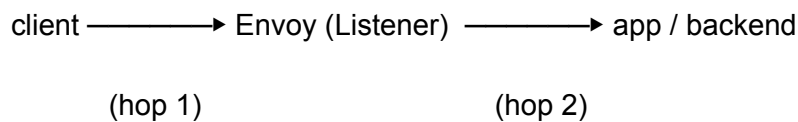
`InboundTrafficPolicy` is meant to be the general-purpose home for this class of configuration going forward. This doc covers the resource itself: its scope, targeting model, and cross-cutting behavior. Features have their own dedicated design doc.

Design Overview

A new CRD of kind `InboundTrafficPolicy` follows the GEP-713 policy attachment model, using the same `selector` + `targetRefs` targeting as `TrafficExtension` and `Telemetry`.

Scope: Hop 1 vs. Hop 2

Every proxy in the mesh sits between two hops relative to itself: the traffic arriving at it, and the traffic it sends onward.



`DestinationRule` has always owned hop 2. For outbound traffic to any destination configuring the proxy's own connection to whatever it's calling. No resource is currently scoped exclusively to these hop 1 concerns, the traffic arriving at a proxy, before any cluster is even chosen. See appendix for an enumerated general list.

`InboundTrafficPolicy` controls hop 1 only, on every surface it targets:

- **Sidecar:** the listener (`virtualInbound`) is the workload's whole ingress surface; the proxy is topologically inseparable from one app instance. `InboundTrafficPolicy` configures this listener.
- **Gateway:** same idea; listeners are the whole ingress surface, with no separate outbound-classified listener the way sidecar has.
- **Waypoint:** there's no per-target listener at all. `main_internal` is one shared listener, and the policy resolves to exactly one **filter chain** inside it, alongside dozens of other unrelated services' filter chains in that same listener. Note this implies inbound to the waypoint which is distinct from inbound to the application app.

`VirtualService` doesn't sit in either hop, it sits at the boundary between them. Its job is deciding *which* hop-2 destination a hop-1-arrived request goes to (plus route-level dispatch behavior for that choice: retries, timeout, fault injection, mirroring, header rewrite).

API Definition

```
// InboundTrafficPolicy configures inbound/server-side listener
// settings for matched workloads. It is the complement to
// DestinationRule, which covers outbound/client-side behavior.
message InboundTrafficPolicy {
  // Selector targets workloads by label.
  // Mutually exclusive with targetRefs.
```

```

WorkloadSelector selector = 1;

// TargetRefs targets resources by reference (Gateway, Service).
// Mutually exclusive with selector.
repeated PolicyTargetReference target_refs = 2;

// HTTP filter settings.
HttpSettings http = 3;

// TCP/connection-level settings.
TcpSettings tcp = 4;
}

message HttpSettings {
  // See the Compression design doc.
  CompressionSettings compression = 1;

  // See the Local Rate Limiting design doc.
  LocalRateLimitSettings local_rate_limit = 2;

  // See the Global Rate Limiting design doc.
  GlobalRateLimitSettings global_rate_limit = 3;
}

message TcpSettings {
  // See the Connection Configuration design doc.
}

```

Port Targeting

InboundTrafficPolicy resolves to one target (a workload, a Service, or a Gateway), but each target can expose multiple ports, and the API needs to account for that.

DestinationRule has the same problem on hop 2, and solves it with a top-level setting overridden per port by a **repeated PortTrafficPolicy** field.

A port-level override doesn't inherit from the top-level setting: any field left unset in the port-level entry falls back straight to Envoy/Istio's own built-in default, not to whatever the top-level setting configured. **InboundTrafficPolicy** follows the same semantics for **port_level_settings**.

```

message InboundTrafficPolicy {
  WorkloadSelector selector = 1;

```

```

    repeated PolicyTargetReference target_refs = 2;
    HttpSettings http = 3;
    TcpSettings tcp = 4;

    // Port-specific overrides. Fields omitted fall back
    // to defaults, not to the http/tcp settings above.
    repeated PortTrafficPolicy port_level_settings = 5;
}

message PortTrafficPolicy {
    PortSelector port = 1;
    HttpSettings http = 2;
    TcpSettings tcp = 3;
}

```

This plugs directly into the per-port/per-listener loop every surface already has: building chain N for port P, check `port_level_settings` for an entry matching P, and layer it over the defaults per the semantics above.

Kube Gateway API Scope

For Kube Gateway API, `InboundTrafficPolicy` targets a `Gateway` and applies to all of its listeners by default, with per-listener scoping via `port_level_settings` as described in [Port Targeting](#) above.

`HTTPRoute` targeting is not supported. `InboundTrafficPolicy` is a listener-scoped resource therefore most of its surface has no meaningful route-level mapping. Where a specific feature *can* be expressed per route, each feature's own design doc covers how ownership can be correctly scoped.

Conflict Resolution and Status

Policy attachment makes overlap inevitable: multiple `InboundTrafficPolicy` resources can select the same proxy. Resolution follows the precedence Istio already uses for `Telemetry`, `Sidecar`, and `PeerAuthentication`, from most to least specific:

1. `targetRefs` (binds a specific resource).
2. `selector` with `matchLabels`.
3. Namespace-wide policy (empty `selector`) in the workload's namespace.
4. Mesh-wide default in the root namespace (`istio-system`).

The single most specific policy wins and applies in full; policies are not merged (which would produce surprising cross-author interactions, e.g. concatenated compressor lists). Ties at the

same precedence level are broken by oldest `creationTimestamp`, then by name, matching existing Istio convention.

`InboundTrafficPolicy` carries a `status` subresource reporting GEP-713-style conditions per target:

- `Accepted`: the policy attached to at least one proxy.
- `Conflicted` / `Overridden`: a more specific policy won for some or all targets; the winning policy is named so the author can tell their config is shadowed.
- `TargetNotFound`: the referenced Gateway/Service/selector matched nothing, or targets a Service with no waypoint bound via istio.io/use-waypoint.

Alternatives Considered

Keep Status Quo

Not implementing this API and leaving all of this configuration on `EnvoyFilter`. The [Background](#) section above provides justification for why this approach should be improved.

Split L4 and L7 CRDs

Rather than one `InboundTrafficPolicy` spanning both layers, split inbound configuration along the L4/L7 boundary into two resources, each at a coherent altitude:

- `InboundListenerPolicy` (L3/L4): listener and connection settings: `maxConnections`, TCP keepalive, connection limits, listener-level timeouts. Targets a `Gateway` or a workload `selector`.
- `InboundHTTPPolicy` (L7): HTTP filter settings: compression, local rate limiting, buffer, etc. Because it is purely L7, it can target route-level resources (`HTTPRoute`, `VirtualService`) in addition to `Gateway` and `Service`.

```
# L4
apiVersion: networking.istio.io/v1alpha1
kind: InboundListenerPolicy
metadata:
  name: ingress-listener
  namespace: istio-system
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: ingressgateway
  tcp:
    maxConnections: 10000
```

```

    keepalive:
      time: 7200s
---
# L7
apiVersion: networking.istio.io/v1alpha1
kind: InboundHTTPPolicy
metadata:
  name: ingress-compression
  namespace: istio-system
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: ingressgateway
  compression:
    compressors:
    - brotli:
        quality: 4
        minLength: 1024
    - gzip: {}

```

Both proposed CRDs here would still be entirely hop-1, in the terms established in [Scope: Hop 1 vs. Hop 2](#); the L4/L7 split is a separate axis (protocol layer within hop 1), not the same distinction as hop 1 vs. hop 2.

The advantages over the single umbrella are that each resource has a coherent altitude, conflict resolution happens within a single layer (an HTTP policy can never shadow an L4 policy, removing the cross-section merge question entirely), and the L7 resource gains natural per-route targeting: `HTTPRoute/VirtualService` map cleanly onto Envoy's per-route filter overrides, closing the per-route gap the umbrella defers.

The drawbacks:

- **Two CRDs instead of one.** The TOC has a standing preference for fewer CRDs: each new kind is API surface to document, version, validate, and graduate, and adds to the conceptual load a user carries when reasoning about what configures their proxy. Two inbound-policy kinds also invite confusion with the existing outbound `DestinationRule`, giving three "traffic policy" resources to disambiguate.
- **A fully-configured inbound listener may require both resources.** An operator who wants, say, connection limits *and* compression on the same gateway must author and reconcile two objects rather than one, duplicating the targeting boilerplate (`targetRefs/selector` repeated in each) and creating two independent status

surfaces to check. It also splits ownership awkwardly when L4 and L7 settings are conceptually a single intent for one listener.

- **The L4/L7 boundary is not always obvious.** Some settings don't fall cleanly on one side. Connection-level timeouts (idle timeout, max connection duration, drain timeout) apply to any connection (raw TCP or HTTP), so they read as L4, yet operators reason about them alongside HTTP behavior. HTTP/2 tuning (max concurrent streams, stream/connection window, header table size) is enforced at the connection layer but is meaningless outside HTTP: L4 in mechanism, L7 in meaning.

Appendix

Hop 1 Configuration

Per [Scope: Hop 1 vs. Hop 2](#), `InboundTrafficPolicy` claims hop 1: the inbound listener/filter chain traffic arrives at, before any cluster is chosen. This is the inventory of what already configures hop 1 today, across all four surfaces, so `InboundTrafficPolicy`'s own filters compose with what's already there rather than silently conflicting with it.

Resource	Mechanism	Surfaces	Notes
<code>PeerAuthentication</code>	mTLS mode → filter-chain match + <code>DownstreamTlsContext</code> (<code>policy_appplier.go:113-140</code>)	Sidecar only (<code>listener_inbound.go:227</code>)	Gateway never consults it (<code>Server.tls</code> governs its TLS instead); waypoint's real arrival listener (<code>buildConnectTerminateListener</code> , <code>listener_waypoint.go:316-332</code>) hardcodes <code>RequireClientCertificate: true</code> unconditionally, bypassing <code>PeerAuthentication</code> entirely
<code>AuthorizationPolicy</code>	RBAC filter, HTTP + TCP variants, via the shared <code>buildCompleteNetworkFilters</code> (<code>networkfil</code>	All three surfaces, symmetric	None

Resource	Mechanism	Surfaces	Notes
	<code>ter.go:203-232)</code>		
RequestAuthentication	JWT HTTP filter (<code>authentication.go:97-108</code>)	Sidecar/gateway (<code>listener_builder.go:472</code>) + waypoint (<code>listener_waypoint.go:898</code>)	HTTP-only, no TCP equivalent
Telemetry	Access-log/stats filters	All three surfaces, symmetric	None
WasmPlugin / TrafficExtension	Fixed anchors (AUTHN/AUTHZ/STATS/UNSPECIFIED) via <code>PopAppendHTTPTrafficExtension</code>	Sidecar/gateway (<code>listener_builder.go:468-476</code>) + waypoint (<code>listener_waypoint.go:893-899</code>)	Unresolved <code>ext-authz</code> double-ordering TODO, verbatim identical comment on both surfaces
EnvoyFilter	<code>PatchContext</code> : <code>SIDECAR_INBOUND</code> , <code>GATEWAY</code> , <code>WAYPOINT</code> (<code>envoy_filter.proto:508, 514, 518</code>)	<code>SIDECAR_INBOUND</code> / <code>GATEWAY</code> are real; <code>WAYPOINT</code> is schema-valid and admission-accepted but never wired into any patch-application code	Passes validation, silently patches nothing: worse than an outright rejection
Sidecar CR (<code>Ingress[]</code>)	<code>captureMode</code> (bind vs. iptables capture), <code>tls</code> (<code>ENABLE_TLS_ON_SIDECAR_INGRESS</code> , default off)	Sidecar only	<code>tls</code> is inert for most users, gated behind a feature flag that defaults off
Gateway resource	<code>Server.tls</code> termination	Gateway / Kube Gateway API	Not a conflict; sequential: TLS terminates first,

Resource	Mechanism	Surfaces	Notes
			<code>InboundTrafficPolicy</code> 's filters run on the plaintext stream after
<code>ProxyConfig/Mesh Config</code> (beyond <code>gatewayTopology</code>).	Drain duration, termination drain duration, hold-application-until-proxy-starts	None touch inbound listener filter content	No overlap: process lifecycle/drain sequencing, not listener config

Inbound Traffic Policy: Compression

Shared with Istio Community

	
Owner: Liam White Working Group: Networking	Status: WIP In Review Approved Obsolete Created: 2025-08-26 Approvers:

Overview

Goals

- Allow HTTP response compression to be configured without the **EnvoyFilter** API to provide better ergonomics and guarantees during upgrades.
- Support compression across all relevant Istio data planes: Istio Gateway, sidecars, waypoints, and Kube Gateway API.

Non-Goals

- Client-side (outbound) **request-body** compression, and automatic propagation of compression config to client sidecars. Client-side response *decompression* for transparent east-west compression is in scope, but only as an explicit **DestinationRule** opt-in (see [Sidecar Outbound](#)).

Background

Istio has no native API for HTTP response compression. Users who want to compress traffic to reduce bandwidth between ingress gateways and external clients, or between sidecars within the mesh, must use an **EnvoyFilter** and write raw Envoy configuration directly. This has been requested since 2019 ([#18443](#)).

No other CNCF gateway or mesh implementation has proposed compression upstream to the Gateway API spec. Every existing ingress controller implementation has gone implementation-specific (see [Prior Art](#)).

Design Overview

API Definition

```
message HttpSettings {  
  CompressionSettings compression = 1;  
}
```

```

message CompressionSettings {
    // Ordered list of compressors. The first algorithm matching
    // the client's Accept-Encoding preference is applied.
    repeated Compressor compressors = 1;

    // Skip compression for responses carrying a strong ETag
    // so the validator is preserved for conditional requests.
    // Default: false (matches Envoy).
    google.protobuf.BoolValue disable_on_etag = 2;

    // Strip the Accept-Encoding header from the request before it
    // reaches the application, so the upstream always serves
    // uncompressed and Istio owns compression end to end.
    // Default: true.
    google.protobuf.BoolValue remove_accept_encoding_header = 3;
}

message Compressor {
    // Minimum response body size in bytes to compress.
    // Default: 30 (matches Envoy).
    google.protobuf.UInt32Value min_content_length = 1;

    // Content-Type values eligible for compression.
    // If empty, Envoy's default set of compressible types applies:
    // application/javascript, application/json, application/xhtml+xml,
    // image/svg+xml, text/css, text/html, text/plain, text/xml.
    repeated string content_type = 2;

    oneof compressor_config {
        GzipCompressor gzip = 3;
        BrotliCompressor brotli = 4;
        ZstdCompressor zstd = 5;
    }
}

message GzipCompressor {
    // Compression level (1-9). Default: 6.
    google.protobuf.UInt32Value compression_level = 1;

    // Zlib window bits (9-15). Default: 12.
    google.protobuf.UInt32Value window_bits = 2;
}

```

```

// Zlib memory level (1-9). Default: 5.
google.protobuf.UInt32Value memory_level = 3;

// Output buffer size in bytes. Default: 4096.
google.protobuf.UInt32Value chunk_size = 4;
}

message BrotliCompressor {
  // Quality level (0-11). Default: 11 (matches Envoy).
  google.protobuf.UInt32Value quality = 1;

  // Sliding window size in bits (10-24). Default: 18.
  google.protobuf.UInt32Value window_bits = 2;

  // Input block size in bits (16-24). Default: 24.
  google.protobuf.UInt32Value input_block_bits = 3;

  // Output buffer size in bytes. Default: 4096.
  google.protobuf.UInt32Value chunk_size = 4;
}

message ZstdCompressor {
  // Compression level (1-22). Default: 3.
  google.protobuf.UInt32Value compression_level = 1;

  // Output buffer size in bytes. Default: 4096.
  google.protobuf.UInt32Value chunk_size = 2;
}

```

Sidecar Outbound

For codec-aware clients, response compression needs no client-side configuration: the server's **InboundTrafficPolicy** compresses, and because Envoy's compressor gates on the request's **Accept-Encoding**, a client that advertises gzip receives the compressed response and decompresses it itself. The outbound sidecar does nothing and leaves decompression to the client.

Transparent east-west compression (where the client application does *not* advertise a codec but the response should still be compressed on the wire to save bandwidth) is in scope but **opt-in**, configured explicitly on the client's **DestinationRule**; it is not propagated automatically from the server policy. The outbound sidecar then advertises **Accept-Encoding** upstream and decompresses the response before the application sees it:

```
apiVersion: networking.istio.io/v1
```

```
kind: DestinationRule
metadata:
  name: reviews-client-compression
  namespace: default
spec:
  host: reviews.default.svc.cluster.local
  trafficPolicy:
    compression:
      response:
        decompress:
          enabled: true
```

Request-body compression (the client compressing the request it sends upstream) is **out of scope**, but is left as a future possibility by the proposed api schema.

Every mechanism in this section (codec-aware default and opt-in decompression alike) configures the *calling* client's own hop 2 (that client's Envoy → this service) via `DestinationRule`, not anything `InboundTrafficPolicy` touches, inline with the resource hop 1 scoping.

Compression Semantics

The compressor list is ordered by preference, not merged. Istio installs one Envoy compressor filter per configured algorithm in the HTTP connection manager chain, in list order. For a given response Envoy walks the chain and the first algorithm in list order that both matches the client's `Accept-Encoding` (respecting q-values) and passes its own `content_type/min_content_length` checks wins; a filter whose checks fail falls through to the next, and a response is compressed **at most once**. Because those checks are per-compressor, the threshold and eligible types that apply are the winning algorithm's. So two algorithms can carry different `content_type` sets and the chosen one depends on both negotiation and per-filter eligibility.

The following behaviors are handled automatically and are not separately configurable:

- **Vary: Accept-Encoding** is added to every compressible response so shared caches key correctly on the negotiated encoding.
- **Double-compression avoidance:** responses that already carry a `Content-Encoding` are passed through untouched.
- **Uncompressed by default:** responses below `min_content_length`, with a non-eligible `content_type`, or whose client sent no matching `Accept-Encoding`, are served as-is.

Configurable cross-cutting behavior lives on `CompressionSettings`:

- **disable_on_etag** (default **false**): skip compression for responses with a strong ETag so conditional-request validators are preserved.
- **remove_accept_encoding_header** (default **true**): strip **Accept-Encoding** before the request reaches the application so the upstream always serves uncompressed and Istio owns compression.

We deliberately do **not** expose a **defaultEncoding** for **Accept-Encoding: ***; the first configured compressor is used.

Security Considerations

Making compression a one-line, mesh-wide toggle lowers the bar to enabling it everywhere, however there are security tradeoffs in doing so:

- **Compression side-channels.** Compressing a response that mixes a secret (e.g. CSRF token) with attacker-reflected input leaks the secret via response size. This is inherent in all compression and is not specific to Istio.
- **CPU / DoS.** High compression levels are CPU-intensive and can be a footgun under load. Defaults follow Envoy, and two of those defaults are aggressive: brotli quality is **11** (the maximum), and **min_content_length** is **30 bytes**. Operators on high-throughput listeners should consider lowering the brotli quality and raising **min_content_length** (zstd and gzip max levels are likewise expensive). We mirror Envoy's defaults for consistency, but should call this tuning out prominently in user-facing docs.

Alternatives Considered

CompressionPolicy

A compression-specific CRD following the same GEP-713 policy attachment model. Scope is deliberately narrow: only compression configuration.

```
apiVersion: networking.istio.io/v1alpha1
kind: CompressionPolicy
metadata:
  name: ingress-compression
  namespace: istio-system
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: ingressgateway
```

```

compressors:
- brotli:
    quality: 4
    minContentLength: 1024
- gzip: {}

```

All surfaces are covered under a single resource and a single API version. The downside is that future built-in HTTP filter needs (local rate limiting, buffer, etc.) each require their own additional CRD rather than being consolidated under a shared inbound listener policy resource, the same drawback the [Split L4 and L7 CRDs](#) alternative has, one level narrower.

Upstream Gateway API Extension + VirtualService Bridge

Propose a GEP to add compression as a built-in feature of the Kubernetes Gateway API spec (either as a standardised `HTTPRoute` filter type or as a policy kind).

Kube Gateway API path (if GEP is accepted):

```

apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
spec:
  rules:
  - filters:
    - type: ResponseCompression
      responseCompression:
        algorithms: [Brotli, Gzip]
        minContentLength: 1024
    backendRefs:
    - name: api-service
      port: 8080

```

Istio-specific path (VirtualService, covers sidecar + Istio Gateway):

```

apiVersion: networking.istio.io/v1
kind: VirtualService
spec:
  http:
  - route:
    - destination:
        host: api-service
      compression:
        compressors:
        - gzip: {}

```

Waypoint (ambient) coverage depends on GAMMA. If the GEP is scoped to include GAMMA-style routing (HTTPRoute with `parentRef: Service`), waypoints are covered by the same mechanism as the Kube Gateway API path.

The main drawbacks are the uncertainty of GEP acceptance, timelines and the split API surface: two different resources are required to cover the full set of Istio data planes.

Appendix

Prior Art

No implementation has proposed compression upstream to the Gateway API spec. All existing implementations went implementation-specific.

Envoy Gateway: BackendTrafficPolicy + ClientTrafficPolicy ([compression docs](#))

Envoy Gateway splits inbound and outbound concerns across two CRDs, the same boundary that separates `InboundTrafficPolicy` from `DestinationRule`:

- BackendTrafficPolicy (added v1.3, January 2025): outbound/upstream settings including compression. Targets Gateway, HTTPRoute, GRPCRoute, TCPRoute, UDPRoute, or TLSRoute via targetRefs. The compressor field takes an ordered list of algorithms with per-algorithm tuning and a minContentLength threshold.
- ClientTrafficPolicy: inbound/listener settings including TCP keepalive, connection limits, and HTTP/1–HTTP/2 settings on the client-facing listener, the same class of arrival-side connection configuration `InboundTrafficPolicy`'s `tcp/http` fields cover more broadly.

kgateway: TrafficPolicy CRD ([docs](#))

kgateway uses a TrafficPolicy CRD that bundles auth, rate limiting, CORS, fault injection, circuit breaking, and compression under one resource. The compression piece lives at `spec.compression` and explicitly separates response compression from request decompression.

Contour: global config file ([docs](#))

A single `compression.algorithm` field in Contour's static config file. Supports gzip, brotli, zstd, or disabled. Applied globally to all listeners; no per-route or per-listener control. Simpler but not comparable in scope to Istio's requirements.

Traefik: Middleware CRD ([docs](#))

Traefik exposes compression via a Middleware resource referenced by name from an IngressRoute rule. Structurally similar to ExtensionRef: a separate resource holding the config, referenced inline from the routing resource. Per-route precision is native since each route chooses which middlewares to apply. `compress: {}` is valid with all defaults; explicit fields are `encodings` (priority-ordered list, default: gzip/br/zstd), `minResponseBodyBytes` (default:

1024), `includedContentTypes/excludedContentTypes`, and `defaultEncoding` (fallback when `Accept-Encoding: *`).

nginx Ingress Controller: ConfigMap ([docs](#))

Compression is configured entirely via a global ConfigMap; there is no per-route or per-Ingress control. Both gzip and brotli are supported. Key fields: `use-gzip` (default: false), `gzip-level` (default: 1), `gzip-min-length` (default: 256 bytes), `gzip-types`; `enable-brotli` (default: false), `brotli-level` (default: 4), `brotli-min-length` (default: 20 bytes), `brotli-types`.

Summary

Implementation	Approach	Granularity
Envoy Gateway	Two CRDs (BackendTrafficPolicy for compression, ClientTrafficPolicy for TCP inbound), policy attachment	Gateway, HTTPRoute, or other route kinds
kgateway	Single CRD (TrafficPolicy), policy attachment	HTTPRoute
Traefik	Dedicated CRD (Middleware), referenced inline from route	Per-route rule
Contour	Global config file	Global only
nginx Ingress	Global ConfigMap	Global only

Inbound Traffic Policy: Connection Configuration

Shared with Istio Community

	
Owner: Liam White Cameron Garrison Working Group: Networking	Status: WIP In Review Approved Obsolete Created: 2025-08-26 Approvers:

Overview

Goals

- Give operators a declarative, target-specific way to configure connection-level admission and lifecycle limits on the arrival side (hop 1). Existing APIs either configure hop 2 (`DestinationRule`'s `direction == inbound` gate, `Sidecar.InboundConnectionPool/Ingress[.connectionPool]`), provide only a proxy-wide HTTP baseline (`ProxyConfig.ConnectionSettings`), or do not exist as a declarative API (TCP idle timeout is sourced from undocumented proxy metadata, `IDLE_TIMEOUT`, `pkg/model/proxy.go:277`).
- Cover both L4 (raw TCP, protocol-agnostic) and L7 (HTTP/H2-specific) settings, uniformly across sidecar, gateway, waypoint, and Kube Gateway API wherever the underlying Envoy primitive allows it.
- Be explicit about the one field where "uniform" breaks down (`tcpKeepalive` on waypoint) rather than silently under- or over-scoping it.

Non-Goals

- **Migrating or deprecating hop-2 fields.** `DestinationRule.trafficPolicy.connectionPool`, `Sidecar.InboundConnectionPool`, and `Sidecar.Ingress[.connectionPool]` keep meaning exactly what they mean today. This doc only adds new hop-1 fields; see [Relationship to Sidecar/DestinationRule's ConnectionPoolSettings](#).
- **The genuinely hop-2-only subset.** `connectTimeout`, request concurrency (`http2MaxRequests/http1MaxPendingRequests`), `maxRetries`, `retryBudget`, `useClientProtocol` have no possible hop-1 equivalent: they only mean something at the point Envoy dispatches to an upstream. Not attempted here.
- **Per-route granularity.** Doesn't apply structurally: these are connection-level settings, and one TCP connection (especially with HTTP keep-alive) can carry many requests across many routes.
- **Client-identity handling at the edge.** Trusted-proxy count (`numTrustedProxies`), client-cert forwarding (`forwardClientCertDetails`), and PROXY protocol are out of

scope for now. They're currently configured Gateway-only, mesh-wide/per-proxy, via `ProxyConfig.gatewayTopology`, with no equivalent on sidecar or waypoint. Bringing them into `InboundTrafficPolicy` is architecturally the same class of thing as the rest of this doc.

Background

Two Envoy primitives back these fields, depending on whether the target listener speaks raw L4 or HTTP:

- **L4** (`TcpProxy`, `connection_limit` network filter): `TcpProxy` has its own `idle_timeout` (`tcp_proxy.proto:284`) and `max_downstream_connection_duration` (`:321`), both protocol-agnostic; `connection_limit` (`connection_limit.proto:31`) enforces `max_connections` as a plain network filter.
- **L7** (`HttpConnectionManager`): `idleTimeout/maxConnectionDuration/maxRequestsPerConnection` live on `common_http_protocol_options` (`http_connection_manager.proto:522`; `max_requests_per_connection` specifically at `protocol.proto:409`); HTTP/2 keepalive on `http2_protocol_options.connection_keepalive` (`:548, protocol.proto`). All of these are sub-fields of `HttpConnectionManager` itself, built independently per filter chain on every surface, including waypoint, where each targeted Service+port gets its own freshly-constructed HCM (`listener_waypoint.go:911-963`).
- **TCP keepalive** (`SO_KEEPALIVE` and friends) only exists at `Listener.socket_options` (`listener.proto:331`); `FilterChain` has no socket-options field of its own (`listener_components.proto:201-250`), because these apply at `accept()`-time, before filter-chain matching happens.

Relationship to `ProxyConfig.ConnectionSettings`

`ProxyConfig.ConnectionSettings` and `InboundTrafficPolicy` are complimentary.

`ConnectionSettings` provides a proxy-wide baseline, including settings that also affect sidecar outbound traffic. `InboundTrafficPolicy` configures target-specific downstream inbound behavior.

A field belongs in `InboundTrafficPolicy` when it configures an inbound HCM, TCP proxy, or filter chain and can safely differ by target. Proxy-wide resource limits, listener-socket settings and upstream cluster settings remain in CS.

The two APIs divide fields like so:

- ITP only: ``maxConnections``, ``maxRequestsPerConnection``, and ``http2Keepalive``. These are new, target-specific inbound capabilities.

Listener-scoped exception: `tcpKeepAlive` may be expressed through ITP only when the attachment covers an entire effective Listener; otherwise it is rejected.

- Both APIs: ``idleTimeout`` and ``maxConnectionDuration``. For HTTP inbound traffic, ITP overrides the proxy-wide CS value. CS remains the baseline and continues to apply sidecar outbound HCMs.
- `ConnectionSettings` only: the EDGE profile, listener and cluster buffer limits, listener and global downstream connection limits, and `httpdraintimeout`. These are outside of ITP's feature scope.

For overlapping inbound HTTP settings, precedence is:

ITP then `ConnectionSettings`, then existing/default istio behavior. For idle timeout, the legacy ``IDLE_TIMEOUT`` metadata is the temporary fallback after CS and before the envoy default. This could be deprecated after ITP is available.

Relationship to Sidecar/DestinationRule's `ConnectionPoolSettings`

`Sidecar.InboundConnectionPool` and per-port `Ingress[ ].ConnectionPool` set connection-pool/circuit-breaker limits on the sidecar's own inbound cluster, the same thing `DestinationRule.trafficPolicy.connectionPool` sets when gated to `direction == inbound`. Almost everything in `ConnectionPoolSettings` is a hop-2 concern, and `InboundTrafficPolicy` doesn't own hop 2, so most of it is out of scope for this CRD entirely, not something this doc touches. Two categories of fields are the exception, hop-1 concerns wearing a hop-2 costume:

- **`maxConnections`**: today enforced as a cluster circuit breaker, rejecting after the downstream connection is already accepted and dispatched to hop 2. What most operators actually want is to protect Envoy's own accept path. Becomes a new, additive hop-1 field (the listener-side `connection_limit` filter).
- **`idleTimeout`, `maxConnectionDuration`, `tcpKeepalive/http2KeepAlive`, `maxRequestsPerConnection`**: each has a real hop-2 declarative field today (on the cluster) and a real Envoy downstream/HCM primitive with no declarative field at all. Someone setting `DestinationRule`'s `idleTimeout` to guard against idle clients is, today, only tuning the loopback connection to the app; the client-facing side is untouched. These become new, additive hop-1 fields, leaving the existing hop-2 fields meaning exactly what they already mean. `maxRequestsPerConnection` becomes two hop-1 fields, not one (`httpMaxRequestsPerConnection/http2MaxRequestsPerConnection`), since the right value differs enough by protocol to be worth splitting; see [API Definition](#).

Everything else in `ConnectionPoolSettings`: `connectTimeout`, request-concurrency (`http2MaxRequests`, `http1MaxPendingRequests`), `maxRetries`, `retryBudget`, `useClientProtocol`, is structurally hop-2-only: a connect timeout and a retry only mean something at the point Envoy dispatches to an upstream. `InboundTrafficPolicy` has nothing to add here. `Sidecar`'s copies remain candidates for deprecation, but only in favor of

DestinationRule's existing inbound-gated equivalents, a consolidation entirely on hop 2, independent of this CRD.

Design Overview

API Definition

```
message InboundTrafficPolicy {
  WorkloadSelector selector = 1;
  repeated PolicyTargetReference target_refs = 2;
  HttpSettings http = 3;
  TcpSettings tcp = 4;
}

message TcpSettings {
  // connection_limit network filter. Protocol-agnostic:
  // applies whether the target speaks HTTP or raw TCP.
  google.protobuf.UInt32Value max_connections = 1;

  // TcpProxy.idle_timeout for L4-only targets; HCM's
  // common_http_protocol_options.idle_timeout for HTTP targets. Same
  // user-facing field, different underlying Envoy config depending
  // on the target's detected protocol.
  google.protobuf.Duration idle_timeout = 2;

  // TcpProxy.max_downstream_connection_duration (L4) or HCM's
  // common_http_protocol_options.max_connection_duration (HTTP).
  Same
  // resolution rule as idle_timeout.
  google.protobuf.Duration max_connection_duration = 3;

  // Listener.socket_options (SO_KEEPALIVE and friends). Rejected at
  // admission time when targeting a waypoint;
  // see the tcpKeepalive Exception.
  TcpKeepalive tcp_keepalive = 4;
}

message TcpKeepalive {
  google.protobuf.UInt32Value probes = 1;
  google.protobuf.Duration time = 2;
  google.protobuf.Duration interval = 3;
}

message HttpSettings {
```

```

CompressionSettings compression = 1;
// ...local_rate_limit, etc. from the other docs

// HCM common_http_protocol_options.max_requests_per_connection.
// Applies to all HTTP protocols handled by this HCM.
google.protobuf.UInt32Value max_requests_per_connection = 10;

// HCM http2_protocol_options.connection_keepalive. HTTP/2-only.
Http2KeepaliveSettings http2_keepalive = 11;
}

message Http2KeepaliveSettings {
  google.protobuf.Duration interval = 1;
  google.protobuf.Duration timeout = 2;
}

```

`idle_timeout` and `max_connection_duration` live on `TcpSettings`, not `HttpSettings`, even though they sometimes compile to an HTTP-specific field. The user-facing concept ("how long can this connection sit idle") is the same regardless of protocol, and asking someone to declare it twice for what's conceptually one knob would be worse ergonomics than the implementation branching on the target's protocol, which Istio already does today when deciding whether to build a `tcp_proxy` or an `HttpConnectionManager` for a given port.

`max_requests_per_connection` applies to every HTTP protocol handled by an HCM. A single HCM can accept both HTTP/1 and HTTP/2, for example through ALPN on an HTTPS Gateway listener, so protocol-specific values cannot be selected after negotiation. Users should choose an HTTP/2-safe value when both protocols are possible .

Per-Surface Mechanics

Field	Sidecar	Gateway / Kube Gateway API	Waypoint
<code>maxConnections</code>	Fine: filter chain-scoped	Fine	Fine: confirmed independent per-(Service, port) <code>FilterChain</code> (<code>buildWaypointInternal</code> , <code>listener_waypoint.go:388-493</code>)

Field	Sidecar	Gateway / Kube Gateway API	Waypoint
<code>idleTimeout</code> , <code>maxConnectionDuration</code>	Fine	Fine	Fine: same per-target <code>FilterChain/HCM</code> independence
<code>maxRequestsPerConnection</code> , <code>http2Keepalive</code>	Fine: HCM sub-field	Fine	Fine: HCM built fresh per target (:911-963)
<code>tcpKeepalive</code>	Fine: listener already belongs to one target	Fine: same	Not fine : see below

Every field except `tcpKeepalive` binds at the filter-chain level, already independently scoped per target on every surface, including waypoint's shared `main_internal` listener.

`tcpKeepalive`

`tcpKeepalive` applies at `Listener` scope in Envoy (raw socket options), because it has to act at `accept()`-time, ahead of filter-chain matching. Setting it via an `InboundTrafficPolicy` that targets a waypoint would mean setting it for the entire `main_internal` listener: every Service sharing that waypoint, not just the one this policy targets.

This leaves up with two options: silently widen the blast radius (apply listener-wide whenever any targeted policy sets it), or reject at admission time.

This design picks the latter: validate that `tcp.tcpKeepalive` is unset whenever `target_refs` resolves to a waypoint, with a clear status condition explaining why. Gateway/Kube Gateway API is unaffected, since it already builds one separate `Listener` per port/server (`buildGatewayListeners`), so its listener already belongs to exactly one target by construction.

`port_level_settings` extends this same conflict to **sidecar**. Sidecar's `virtualInbound` is a single `Listener` (`inboundVirtualListener`, `listener_inbound.go:262`) holding every one of a workload's ports as separate filter chains underneath it, one `Listener`, many ports. `tcpKeepalive` genuinely cannot differ per port there, because there's only one listener object for its socket options to live on, shared by every port. So the same admission-time rejection applies to a `tcpKeepalive` entry inside `port_level_settings` on sidecar, not only on waypoint: whole-target `tcpKeepalive` is fine.

Deprecating the Bootstrap-Metadata Idle Timeout

Today's only declarative-adjacent path to a downstream idle timeout is `ProxyConfig`'s undocumented `IDLE_TIMEOUT` node metadata (`pkg/model/proxy.go:277`), a debugging

escape hatch, not a real API field. Once `tcp.idleTimeout` exists, this doc proposes deprecating that metadata path on a normal timeline (warn, then remove) rather than leaving two ways to set the same thing indefinitely. In the meantime, this resource will override the existing setting.

Alternatives Considered

EnvoyFilter today. For sidecar and gateway, this is a real, working (if imperative, per-workload) option: `EnvoyFilter`'s `SIDECAR_INBOUND/GATEWAY` listener-patching contexts do reach these listeners. For waypoint, it's not an option at all: `listener_waypoint.go` never calls into the `envoyfilter` package for listeners under any context, including `WAYPOINT`, which exists in the proto but is never wired into patch-application code. The status quo is "hand-roll it via `EnvoyFilter` on two of four surfaces, impossible on the third," which motivates a declarative, uniform field instead.

Inbound Traffic Policy: Local Rate Limiting

Shared with Istio Community

	
Owner: Liam White Working Group: Networking	Status: WIP In Review Approved Obsolete Created: 2025-08-26 Approvers:

Goals

- Let operators cap the rate of new connections and/or requests arriving at a proxy's inbound listener (hop 1), per `InboundTrafficPolicy` target, on every surface (sidecar, gateway, waypoint, Kube Gateway API).
- Do it without requiring an external service: this is the "cheap, always-available" tier of rate limiting.
- Get reasonably close to correct aggregate behavior when a target has multiple replicas (HA gateway, autoscaled waypoint), not just "correct per-pod."
- Allow a limit to vary by route (path, method, headers) within a target, not just apply uniformly to the whole listener/filter chain; see [Per-Route Configuration](#).

Non-Goals

- **Exact, globally-consistent counting.** An external rate-limit-service-backed approach exists for cases needing a precise, cross-replica count rather than an approximation; that's a separate filter and a separate tradeoff. Local rate limiting is inherently approximate; the multi-replica sharing mechanism below narrows the approximation, it doesn't eliminate it.
- **Hop 2.** Rate limiting the proxy's own connection to its local app or backend pods.

Background

Envoy has two local rate limit filters relevant here:

`envoy.filters.http.local_ratelimit` (per-request, HTTP-level, descriptor-based) and `envoy.filters.network.local_ratelimit` (per-connection, L4-level). Both default to a token bucket scoped to the single Envoy process evaluating them; not shared across replicas of the same workload.

Envoy has a purpose-built escape hatch for that: the `local_cluster_rate_limit` share-provider mechanism (`ShareProviderManager`, `local_ratelimit_impl.h`), which divides the bucket by $1/N$ live replicas, tracked via a designated "local cluster"'s `membership_total` stat and kept current via `addMemberUpdateCb`. Not wired up anywhere in Istio today: `local_cluster_name` isn't referenced anywhere in `pilot/`.

Design Overview

API Shape

New fields on both existing settings messages, since this is meaningful at both L4 (connection rate) and L7 (request rate, with descriptors):

```
message TcpSettings {
  // ...existing fields (maxConnections, idleTimeout, etc.)
  LocalRateLimitSettings local_rate_limit = 5;
}

message HttpSettings {
  CompressionSettings compression = 1;
  LocalRateLimitSettings local_rate_limit = 2;
}

message LocalRateLimitSettings {
  // Default bucket, applied when no descriptor matches
  // (or always, if no descriptors are configured at all).
  TokenBucket default_bucket = 1;

  // Ordered list of descriptor-based buckets. First match wins.
  // HTTP-only since a raw TCP connection has no request attributes
  // to key a descriptor on.
  repeated RateLimitDescriptor descriptors = 2;

  // If true, evaluate but just emit stats/logs.
  // Maps to Envoy's filter_enabled/filter_enforced runtime split.
  google.protobuf.BoolValue shadow_mode = 3;
}

message TokenBucket {
  uint32 max_tokens = 1;
  uint32 tokens_per_fill = 2;
  google.protobuf.Duration fill_interval = 3;
}

message RateLimitDescriptor {
  // Match against request attributes (header presence/value,
  // remote address, etc.) – deliberately request-attribute matching
  // only, not path/host routing. Reuses the same match vocabulary
  // HTTPMatchRequest already uses elsewhere in Istio, not a new one.
```

```

    repeated DescriptorMatch match = 1;

    TokenBucket bucket = 2;
}

```

Per-Surface Mechanics

Surface	Binds to	Replica sharing needed?
Sidecar	The one <code>virtualInbound</code> listener/HCM for the targeted workload	No: exactly one Envoy per targeted workload by construction, so per-process is already per-workload
Gateway	The gateway's own listener(s)	Yes, if the Gateway's Deployment has >1 replica , common for HA/scaled ingress
Waypoint	The one filter chain + HCM built per targeted Service+port (confirmed independent per target: waypoint builds one <code>FilterChain</code> per (Service, port) in a loop, <code>buildWaypointInternal</code> , <code>listener_waypoint.go:388-493</code> , each with its own freshly-constructed HTTP connection manager, <code>buildWaypointInboundHTTPFilters</code> , <code>listener_waypoint.go:911-963</code>)	Yes, if the waypoint's Deployment has >1 replica , common for HA/scaled waypoints
Kube Gateway API	Same as Gateway	Same as Gateway

The binding itself is de-risked by that same construction: waypoint's per-(Service, port) filter chains each get their own independently-built HCM, so a `local_ratelimit` filter with its own config drops in cleanly per target, no `typed_per_filter_config` gymnastics needed to keep one Service's limit from bleeding into a neighboring one's, despite `main_internal` being a shared listener.

Per-Route Configuration

A real class of user need sits one level narrower than "how much traffic can this listener/filter chain admit": a strict limit on `POST /auth/login`, a loose one on `GET /catalog`, both behind the same target. This doesn't need a route-targeting mechanism at all:

`local_ratelimit` generates its descriptor entries via its own `rate_limits` field (`local_rate_limit.proto:183`), which reuses Envoy's `route.v3.RateLimit.Action` vocabulary, including `HeaderValueMatch`, which matches against `:path` as a pseudo-header exactly like any other header. Paired with `descriptors` (the verbatim-match token-bucket table), this generates a different bucket per path (or method, or any header) entirely within the filter's own config.

Multi-Replica Sharing

Most of the infrastructure this needs already exists upstream, just not for this purpose. Istio has an opt-in proxy metadata flag, `ISTIO_META_ENABLE_SELF_DISCOVERY` (`pilot/cmd/pilot-agent/options/options.go:99-102`, off by default), that injects an EDS cluster named `local_cluster` into the proxy's bootstrap alongside

```
cluster_manager.local_cluster_name: "local_cluster"
```

```
(pkg/bootstrap/config.go:210-211;
```

```
tools/packaging/common/envoy_bootstrap.json:560-589,1106-1109), exactly the bootstrap field local_cluster_rate_limit's ShareProviderManager needs.
```

Membership resolves per-proxy at discovery time to that proxy's own `LocalService` (`eds.go:271-281`; `pkg/model/context.go:630-641`, `SetServiceTargets`), the proxy's own backing Service, i.e. its own sibling replica set for a gateway or waypoint.

- **Gateway / Kube Gateway API / Waypoint:** default `ISTIO_META_ENABLE_SELF_DISCOVERY` to true for these proxy types, unconditionally, rather than toggling it per-policy. The generated `local_ratelimit` filter's `share_provider` points at the already-injected `local_cluster` whenever a target actually sets `local_rate_limit`.
- **Sidecar:** skip entirely, same reasoning as before: always exactly one replica per target by construction, so there is no sharing problem to solve.

Alternatives Considered

EnvoyFilter today. Technically possible to hand-write a `local_ratelimit` patch right now.

Ownership: Target the Service or Workload, Not the Shared Gateway

[Per-Route Configuration](#) covers *matching*: one policy, one author, buckets that vary by path/header. A related but distinct question is *ownership*: how does a team get their own rate limit without editing a shared resource they don't own, when multiple teams publish routes to the same Gateway?

If the goal is "rate-limit my service," target the service or workload directly.

`InboundTrafficPolicy`'s existing `targetRefs/selector` model already provides full ownership isolation for this today, on every surface, with no new mechanism required.

A platform team wanting a rate limit on the *aggregate* of all traffic through a shared Gateway, regardless of which team's route it hits (e.g. "no client IP over 1000 req/s across the whole gateway"). This case stays with the platform team's own `InboundTrafficPolicy` on the `Gateway`.

Inbound Traffic Policy: Global Rate Limiting

Shared with Istio Community

	
Owner: Liam White Working Group: Networking	Status: WIP In Review Approved Obsolete Created: 2025-08-26 Approvers:

Goals

- Let a hop-1 filter get an exact, centrally-coordinated admission decision from an external Rate Limit Service (RLS), for cases where local rate limiting's per-target approximation isn't good enough: hard quota/license enforcement, fairness across an entire tenant's traffic mesh-wide, not just one target's replica set.
- Reuse Istio's existing `ExtensionProvider` indirection (`MeshConfig`) to name the RLS, rather than embedding endpoint details inline in every policy instance.
- Uniform behavior across all four surfaces: this feature has none of local rate limiting's per-surface replica-sharing problem, since the RLS is the single shared source of truth by design.
- Allow descriptors (and therefore the RLS decision itself) to vary by route (path, method, headers) within a target, not just apply uniformly to the whole listener/filter chain: see [Per-Route Configuration](#).

Non-Goals

- **Replacing local rate limiting.** Complementary, not a superset: see [Layering](#) below.
- **Provisioning or managing the RLS server.** BYO, same posture Istio already takes for `ExtensionProvider`-referenced `ext_authz`/tracing backends.

Background

`envoy.filters.http.ratelimit` (and its L4 counterpart, `envoy.filters.network.ratelimit`) makes a synchronous gRPC call to an external RLS per matched request/connection, sending a `domain` string plus a list of descriptor entries; the RLS owns all bucketing/counting state and returns an allow/deny decision.

Design Overview

Referencing the RLS: MeshConfig ExtensionProvider

The RLS endpoint, TLS settings, and connection-level config (a real cluster, effectively) don't belong inline in every `InboundTrafficPolicy` instance: that's exactly the pattern

MeshConfig.ExtensionProvider already exists to avoid, and **Telemetry/AccessLogging/ext_authz** already use it this way. A new provider type:

```
// In MeshConfig.ExtensionProvider
message EnvoyRateLimitProvider {
    string service = 1;           // cluster-resolvable RLS address
    uint32 port = 2;
    google.protobuf.Duration timeout = 3;
    ProviderFailureMode failure_mode = 4;
}
```

```
enum ProviderFailureMode {
    // default: reject traffic if the RLS is unreachable
    PROVIDER_FAILURE_MODE_CLOSED = 0;

    // admit traffic if the RLS is unreachable
    PROVIDER_FAILURE_MODE_OPEN = 1;
}
```

InboundTrafficPolicy then only names the provider, plus policy-specific descriptor rules:

```
message GlobalRateLimitSettings {
    // Name of an EnvoyRateLimitProvider from MeshConfig.ExtensionProvider.
    string provider = 1;

    // Sent to the RLS as the request domain - namespaces this policy's
    // descriptors from every other policy's, in case the same RLS backs
    // multiple InboundTrafficPolicy instances. Defaults to
    // "<namespace>/<name>" of this InboundTrafficPolicy at the top level,
    // or "<namespace>/<name>/<port>" when set inside a port_level_settings
    // entry, so two ports of the same target never collide on the default.
    string domain = 2;

    // same shape as the local rate limit
    repeated RateLimitDescriptor descriptors = 3;

    // Overrides the provider's failure_mode for this target only.
    FailureMode failure_mode = 4;
}

enum FailureMode {
    FAILURE_MODE_UNSPECIFIED = 0; // defaults to provider
    FAILURE_MODE_PROVIDER = 1;    // explicitly set provider
    FAILURE_MODE_OPEN = 2;        // admit traffic if the RLS is unreachable
    FAILURE_MODE_CLOSED = 3;      // reject traffic if the RLS is unreachable
}
```

This mirrors `Telemetry`'s existing `provider` string-name indirection almost exactly; no new pattern introduced.

Descriptors and Domain

`RateLimitDescriptor` entries match against request attributes: header presence/value, remote address, and path/method via header-style matching against pseudo-headers like `:path`. Each entry pairs a `match` list with whatever descriptor key/value the RLS should receive for a matching request, so a single policy can send different descriptors to the RLS depending on which entry matched first. `domain` exists because a single RLS deployment commonly backs many independent policies (per Envoy's RLS protocol, `domain` is how the server picks the right rule set); defaulting it to the policy's own `<namespace>/<name>` avoids yet another required field while still keeping instances from colliding by default.

When configured via `port_level_settings` is in the picture: two different ports of the same target could each carry their own `GlobalRateLimitSettings`. In this scenario the default would be set to `<namespace>/<name>/<port>` instead, so two ports never collide without the user having to know to set `domain` explicitly.

Layering with Local Rate Limiting

The two are meant to compose, not compete: a cheap local rate limit as a first-pass filter to shed obviously excessive traffic before paying the latency and cost of a synchronous RLS round-trip, with global rate limiting as the precise backstop behind it. Both `LocalRateLimitSettings` and `GlobalRateLimitSettings` can be set on the same `InboundTrafficPolicy` target simultaneously; ordering in the generated filter chain should put local before global for exactly this reason (fail fast and cheap before paying for the expensive, exact check).

Security Considerations

mTLS to the RLS cluster. Since the provider resolves to a real Istio-managed cluster, it should get the same automatic mesh mTLS treatment any other in-mesh `ExtensionProvider`-referenced backend gets: no separate TLS configuration surface needed if the RLS itself is in-mesh.

Alternatives Considered

Embedding RLS endpoint config inline in `InboundTrafficPolicy`. Rejected: breaks the `ExtensionProvider` convention every other cross-cutting backend reference already follows in Istio, and turns an RLS endpoint rotation into an N-CRD-instance update instead of a single `MeshConfig` change.

EnvoyFilter today. Technically possible to hand-write a `ratelimit` patch right now, but `EnvoyFilter`'s listener-patching contexts (`SIDECAR_INBOUND`, `GATEWAY`) don't cover waypoint at all: `listener_waypoint.go` never calls into the `envoyfilter` package. So this is available on sidecar/gateway today, imperatively, per-workload, and simply absent on waypoint. Correctly wiring an RLS cluster reference by hand via `EnvoyFilter` is also

considerably more error-prone than a declarative provider reference, motivating a uniform CRD field over asking users to hand-roll this per surface.

Ownership: Target the Service or Workload, Not the Shared Gateway

[Per-Route Configuration](#) covers *matching*: one policy, one author, buckets that vary by path/header. A related but distinct question is *ownership*: how does a team get their own rate limit without editing a shared resource they don't own, when multiple teams publish routes to the same Gateway?

If the goal is "rate-limit my service," target the service or workload directly.

`InboundTrafficPolicy`'s existing `targetRefs/selector` model already provides full ownership isolation for this today, on every surface, with no new mechanism required.

A platform team wanting a rate limit on the *aggregate* of all traffic through a shared Gateway, regardless of which team's route it hits (e.g. "no client IP over 1000 req/s across the whole gateway"). This case stays with the platform team's own `InboundTrafficPolicy` on the `Gateway`.

Inbound Traffic Policy & Compression

Shared with Istio Community

	
Owner: Liam White Working Group: Networking	Status: WIP In Review Approved Obsolete Created: 2025-06-30 Approvers:

TL;DR

Introduce an `InboundTrafficPolicy` CRD to provide native HTTP compression configuration across all Istio data planes without requiring the `EnvoyFilter` API with further expansion planned for edge proxy config and rate limiting.

Overview

Goals

- Allow HTTP response compression to be configured without the `EnvoyFilter` API to provide better ergonomics and guarantees during upgrades.
- Support compression across all relevant Istio data planes: Istio Gateway, sidecars, waypoints, and Kube Gateway API.
- Follow the GEP-713 policy attachment model for consistency with existing Istio APIs such as `Telemetry` and `TrafficExtension`.

Non-Goals

- Ship `InboundTrafficPolicy` with all fields populated in the initial release — the resource will ship with `http.compression` only; additional `http` and `tcp` fields (such as rate limiting) will be added in subsequent releases. Although Edge proxy configuration will likely be added as a fast follow.
- Client-side request body compression via sidecar outbound.

Background

Istio has no native API for HTTP response compression. Users who want to compress traffic to reduce bandwidth between ingress gateways and external clients, or between sidecars within

Shared with Istio Community

the mesh, must use an `EnvoyFilter` and write raw Envoy configuration directly. This has been requested since 2019 ([#18443](#)).

The `EnvoyFilter` API is considered the API of last resort and should be avoided where possible because using it involves breaking through the Istio abstraction and interacting with Envoy APIs directly. This makes it difficult for Istio to provide any guarantees around backwards compatibility — upgrades can cause functionality to break due to Envoy API version changes, filter name changes, filter ordering changes, and so on.

No other CNCF gateway or mesh implementations have proposed compression upstream to the Gateway API spec. All existing ingress controller implementations have gone implementation-specific (see Appendix — Prior Art).

Design Overview

A new CRD of kind `InboundTrafficPolicy` and apiVersion `networking.istio.io/v1alpha1` will be added to the Istio API following the GEP-713 policy attachment model, using the same `selector` + `targetRefs` targeting as `TrafficExtension` and `Telemetry`.

`InboundTrafficPolicy` is the complement to `DestinationRule`, which covers outbound/client-side behavior. `DestinationRule.connectionPool.tcp` configures the proxy's connection pool when connecting to an upstream but has no equivalent for inbound connections from downstream clients. `InboundTrafficPolicy` fills that gap across all data planes, and is explicitly scoped to pave the way for other features currently only supported via `EnvoyFilter` such as global/local rate limiting and edge proxy config.

API Definition

```
// InboundTrafficPolicy configures inbound/server-side listener settings
// for matched workloads. It is the complement to DestinationRule, which
// covers outbound/client-side behavior.

message InboundTrafficPolicy {
  // Selector targets workloads by label. Mutually exclusive with
  targetRefs.
  WorkloadSelector selector = 1;

  // TargetRefs targets resources by reference (Gateway, Service).
  // Mutually exclusive with selector.
  repeated PolicyTargetReference target_refs = 2;

  // HTTP filter settings.
```

```

    HttpSettings http = 3;
}

message HttpSettings {
    CompressionSettings compression = 1;
}

message CompressionSettings {
    // Ordered list of compressors. The first algorithm matching the client's
    // Accept-Encoding preference is applied.
    repeated Compressor compressors = 1;

    // Skip compression for responses carrying a strong ETag so the validator
    is
    // preserved for conditional requests. Default: false (matches Envoy).
    google.protobuf.BoolValue disable_on_etag = 2;

    // Strip the Accept-Encoding header from the request before it reaches
    the
    // application, so the upstream always serves uncompressed and Istio owns
    // compression end to end. Default: false.
    google.protobuf.BoolValue remove_accept_encoding_header = 3;
}

message Compressor {
    // Minimum response body size in bytes to compress. Default: 30 (matches
    Envoy).
    google.protobuf.UInt32Value min_content_length = 1;

    // Content-Type values eligible for compression. If empty, Envoy's
    default
    // set of compressible types applies: application/javascript,
    // application/json, application/xhtml+xml, image/svg+xml, text/css,
    // text/html, text/plain, text/xml.
    repeated string content_type = 2;

    oneof compressor_config {
        GzipCompressor gzip = 3;
        BrotliCompressor brotli = 4;
        ZstdCompressor zstd = 5;
    }
}

message GzipCompressor {

```

```

// Compression level (1-9). Default: 6.
google.protobuf.UInt32Value compression_level = 1;

// Zlib window bits (9-15). Default: 12.
google.protobuf.UInt32Value window_bits = 2;

// Zlib memory level (1-9). Default: 5.
google.protobuf.UInt32Value memory_level = 3;

// Output buffer size in bytes. Default: 4096.
google.protobuf.UInt32Value chunk_size = 4;
}

message BrotliCompressor {
  // Quality level (0-11). Default: 11 (matches Envoy).
  google.protobuf.UInt32Value quality = 1;

  // Sliding window size in bits (10-24). Default: 18.
  google.protobuf.UInt32Value window_bits = 2;

  // Input block size in bits (16-24). Default: 24.
  google.protobuf.UInt32Value input_block_bits = 3;

  // Output buffer size in bytes. Default: 4096.
  google.protobuf.UInt32Value chunk_size = 4;
}

message ZstdCompressor {
  // Compression level (1-22). Default: 3.
  google.protobuf.UInt32Value compression_level = 1;

  // Output buffer size in bytes. Default: 4096.
  google.protobuf.UInt32Value chunk_size = 2;
}

```

Istio Gateway

Targeting an Istio Gateway uses a **selector** on the gateway pods.

```

apiVersion: networking.istio.io/v1alpha1
kind: InboundTrafficPolicy
metadata:
  name: ingress-compression

```

```

namespace: istio-system
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  http:
    compression:
      compressors:
        - brotli:
            quality: 4
            minLength: 1024
        - gzip: {}

```

Sidecar Inbound

Targeting sidecar inbound listeners uses a **selector** on workload pods. The policy applies to the server-side listener, compressing responses sent to downstream clients.

```

apiVersion: networking.istio.io/v1alpha1
kind: InboundTrafficPolicy
metadata:
  name: workload-compression
  namespace: default
spec:
  selector:
    matchLabels:
      app: api-service
  http:
    compression:
      compressors:
        - gzip: {}

```

Sidecar Outbound

For codec-aware clients, response compression needs no client-side configuration: the server's **InboundTrafficPolicy** compresses, and because Envoy's compressor gates on the request's **Accept-Encoding**, a client that advertises gzip receives the compressed response and decompresses it itself. The outbound sidecar does nothing. This is the "compress with gzip, leave decompression to the client" case, and it is the default.

Transparent east-west compression (where the client application does *not* advertise a codec but the response should still be compressed on the wire to save bandwidth) is in scope but **opt-in**, configured explicitly on the client's **DestinationRule**; it is not propagated automatically from

the server policy. The outbound sidecar then advertises **Accept-Encoding** upstream and decompresses the response before the application sees it.

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: reviews-client-compression
  namespace: default
spec:
  host: reviews.default.svc.cluster.local
  trafficPolicy:
    compression:
      response:
        decompress: true
```

Request-body compression (the client compressing the request it sends upstream) is **out of scope** (see [Non-Goals](#)).

Waypoint (Ambient)

Targeting waypoints uses **targetRefs** with **kind: Service**. The policy is applied to the inbound listener filter chains built per-service inside the waypoint.

Compression in ambient is done only by the waypoint, a full L7 Envoy proxy; **ztunnel** is L4-only and cannot compress or negotiate **Accept-Encoding**. Waypoints are not implied by targeting — a Service must be enrolled onto one (via the **istio.io/use-waypoint** label). A policy targeting a Service with no waypoint silently no-ops, the same prerequisite every ambient L7 policy carries.

Admission validation can't catch this due to its single object scoping. istiod instead reports it during reconciliation as a **status** condition (the **TargetNotFound** case in [Conflict Resolution and Status](#)) and an **istioctl analyze** message.

```
apiVersion: networking.istio.io/v1alpha1
kind: InboundTrafficPolicy
metadata:
  name: reviews-compression
  namespace: default
spec:
  targetRefs:
    - kind: Service
      name: reviews
  http:
    compression:
      compressors:
```

```
- gzip: {}
```

Kube Gateway API

For Kube Gateway API, `InboundTrafficPolicy` targets a `Gateway` to apply compression gateway-wide, across all of its listeners. Per-listener scoping is not supported; like per-route targeting it can be added later (additively, via `sectionName`) if demand appears.

`HTTPRoute` targeting is not supported in the initial release. Because `InboundTrafficPolicy` is a listener-scoped resource that will grow listener-level fields (`tcp`), most of its surface has no meaningful route-level mapping. Therefore we default to listener-level compression, which matches the common case and the global/listener model used by Contour and nginx. See the alternatives considered section for discussion on splitting this resource into two: one for L3/L4 and one for L7.

This is a deliberate scoping decision, not a hard limitation. Compression specifically *can* be expressed per route. Per-route compression can therefore be added later by allowing `HTTPRoute` (and Istio `VirtualService`) targeting that maps the `http.compression` block onto per-route filter config, with validation rejecting listener-only fields (e.g. `tcp`) on a route target.

```
apiVersion: networking.istio.io/v1alpha1
kind: InboundTrafficPolicy
metadata:
  name: kube-gateway-compression
  namespace: istio-system
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: prod-gateway
  http:
    compression:
      compressors:
      - gzip: {}
```

Surface Coverage

Surface	Mechanism
Istio Gateway	<code>targetRefs</code> or <code>selector</code> on gateway pods
Sidecar inbound	<code>selector</code> on workload pods
Sidecar outbound	Automatic — propagated from server <code>InboundTrafficPolicy</code>
Waypoint (ambient)	<code>targetRefs: kind: Service</code>
Kube Gateway	<code>targetRefs: kind: Gateway</code> (gateway-wide)

Compression Semantics

The compressor list is ordered by preference, not merged. Istio installs one Envoy compressor filter per configured algorithm in the HTTP connection manager chain, in list order. For a given response Envoy walks the chain and the first algorithm in list order that both matches the client's `Accept-Encoding` (respecting q-values) and passes its own `content_type/min_content_length` checks wins; a filter whose checks fail falls through to the next, and a response is compressed **at most once**. Because those checks are per-compressor, the threshold and eligible types that apply are the winning algorithm's — so two algorithms can carry different `content_type` sets and the chosen one depends on both negotiation and per-filter eligibility.

The following behaviors are handled automatically and are not separately configurable:

- **Vary: Accept-Encoding** is added to every compressible response so shared caches key correctly on the negotiated encoding.
- **Double-compression avoidance** — responses that already carry a `Content-Encoding` are passed through untouched.
- **Uncompressed by default** — responses below `min_content_length`, with a non-eligible `content_type`, or whose client sent no matching `Accept-Encoding`, are served as-is.

Configurable cross-cutting behavior lives on `CompressionSettings`:

- **disable_on_etag** (default `false`) — skip compression for responses with a strong `ETag` so conditional-request validators are preserved.

- **remove_accept_encoding_header** (default `false`) — strip `Accept-Encoding` before the request reaches the application so the upstream always serves uncompressed and Istio owns compression end to end.

We deliberately do **not** expose a `defaultEncoding` for `Accept-Encoding: *`; the first configured compressor is used.

Conflict Resolution and Status

Policy attachment makes overlap inevitable — multiple `InboundTrafficPolicy` resources can select the same proxy. Resolution follows the precedence Istio already uses for `Telemetry`, `Sidecar`, and `PeerAuthentication`, from most to least specific:

1. `targetRefs` (binds a specific resource).
2. `selector` with `matchLabels`.
3. Namespace-wide policy (empty `selector`) in the workload's namespace.
4. Mesh-wide default in the root namespace (`istio-system`).

The single most specific policy wins and applies in full — policies are not merged, so compressor lists from two authors are never concatenated (which would produce surprising negotiation behavior). Ties at the same precedence level are broken by oldest `creationTimestamp`, then by name, matching existing Istio convention.

`InboundTrafficPolicy` carries a `status` subresource reporting GEP-713-style conditions per target:

- **Accepted** — the policy attached to at least one proxy.
- **Conflicted / Overridden** — a more specific policy won for some or all targets; the winning policy is named so the author can tell their config is shadowed.
- **TargetNotFound** — the referenced Gateway/Service/selector matched nothing, or (per the Waypoint section) targets a service with no waypoint on the data path.

Security Considerations

Making compression a one-line, mesh-wide toggle lowers the bar to enabling it everywhere, however there are security tradeoffs in doing so:

- **Compression side-channels.** Compressing a response that mixes a secret (e.g. CSRF token) with attacker-reflected input leaks the secret via response size. This is inherent in all compression and is not specific to Istio.
- **CPU / DoS.** High compression levels are CPU-intensive and can be a footgun under load. Defaults follow Envoy, and two of those defaults are aggressive: brotli quality is **11** (the maximum), and `min_content_length` is **30 bytes**. Operators on high-throughput

listeners should consider lowering the brotli quality and raising `min_content_length` (zstd and gzip max levels are likewise expensive). We mirror Envoy's defaults for consistency, but should call this tuning out prominently in user-facing docs.

Alternatives Considered

Keep Status Quo

Not implementing this API and leaving compression as-is (configured via `EnvoyFilter`). The background section above provides justification for why this approach should be improved.

Split L4 and L7 CRDs

Rather than one `InboundTrafficPolicy` spanning both layers, split inbound configuration along the L4/L7 boundary into two resources, each at a coherent altitude:

- `InboundListenerPolicy` (L3/L4) — listener and connection settings: `maxConnections`, TCP keepalive, connection limits, listener-level timeouts. Targets a `Gateway` or a workload `selector`.
- `InboundHTTPPolicy` (L7) — HTTP filter settings: compression now; local rate limiting, buffer, etc. later. Because it is purely L7, it can target route-level resources (`HTTPRoute`, `VirtualService`) in addition to `Gateway` and `Service`.

```
# L4
apiVersion: networking.istio.io/v1alpha1
kind: InboundListenerPolicy
metadata:
  name: ingress-listener
  namespace: istio-system
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: ingressgateway
  tcp:
    maxConnections: 10000
    keepalive:
      time: 7200s
---
# L7
apiVersion: networking.istio.io/v1alpha1
kind: InboundHTTPPolicy
metadata:
  name: ingress-compression
```

```

namespace: istio-system
spec:
  targetRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: ingressgateway
  compression:
    compressors:
    - brotli:
        quality: 4
        minLength: 1024
    - gzip: {}

```

The advantages over the single umbrella are that each resource has a coherent altitude, conflict resolution happens within a single layer (an HTTP policy can never shadow an L4 policy, removing the cross-section merge question entirely), and the L7 resource gains natural per-route targeting — [HTTPRoute/VirtualService](#) map cleanly onto Envoy's per-route filter overrides, closing the per-route gap the umbrella defers.

The drawbacks:

- **Two CRDs instead of one.** The TOC has a standing preference for fewer CRDs — each new kind is API surface to document, version, validate, and graduate, and adds to the conceptual load a user carries when reasoning about what configures their proxy. Two inbound-policy kinds also invite confusion with the existing outbound [DestinationRule](#), giving three "traffic policy" resources to disambiguate.
- **The L4/L7 boundary is not always obvious.** Some settings don't fall cleanly on one side. Connection-level timeouts (idle timeout, max connection duration, drain timeout) apply to any connection raw TCP or HTTP, so they read as L4, yet operators reason about them alongside HTTP behavior. HTTP/2 tuning (max concurrent streams, stream/connection window, header table size) is enforced at the connection layer but is meaningless outside HTTP — L4 in mechanism, L7 in meaning.

The reason a single resource is proposed is primarily due to the fact that these drawbacks cannot be mitigated whereas validation and admission policies could be added to prevent users from using invalid fields to target the wrong resources.

CompressionPolicy

A compression-specific CRD following the same GEP-713 policy attachment model. Scope is deliberately narrow — only compression configuration.

```

apiVersion: networking.istio.io/v1alpha1

```

```

kind: CompressionPolicy
metadata:
  name: ingress-compression
  namespace: istio-system
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: Gateway
      name: ingressgateway
  compressors:
    - brotli:
        quality: 4
        minContentLength: 1024
    - gzip: {}

```

All surfaces are covered under a single resource and a single API version. The downside is that future built-in HTTP filter needs (local rate limiting, buffer, etc.) each require their own additional CRD rather than being consolidated under a shared inbound listener policy resource.

Upstream Gateway API Extension + VirtualService Bridge

Propose a GEP to add compression as a built-in feature of the Kubernetes Gateway API spec — either as a standardised [HTTPRoute](#) filter type or as a policy kind — and use [VirtualService](#) for the Istio-specific (sidecar + Istio Gateway) path.

Kube Gateway API path (if GEP is accepted):

```

apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
spec:
  rules:
    - filters:
        - type: ResponseCompression
          responseCompression:
            algorithms: [Brotli, Gzip]
            minContentLength: 1024
      backendRefs:
        - name: api-service
          port: 8080

```

Istio-specific path (VirtualService, covers sidecar + Istio Gateway):

```

apiVersion: networking.istio.io/v1
kind: VirtualService

```

```
spec:
  http:
  - route:
    - destination:
        host: api-service
    compression:
      compressors:
      - gzip: {}
```

Waypoint (ambient) coverage depends on GAMMA. If the GEP is scoped to include GAMMA-style routing (HTTPRoute with `parentRef: Service`), waypoints are covered by the same mechanism as the Kube Gateway API path.

The main drawbacks are the uncertainty of GEP acceptance, timelines and the split API surface — two different resources are required to cover the full set of Istio data planes.

Appendix

Prior Art

No implementation has proposed compression upstream to the Gateway API spec. All existing implementations went implementation-specific.

Envoy Gateway — BackendTrafficPolicy + ClientTrafficPolicy ([compression docs](#))

Envoy Gateway splits inbound and outbound concerns across two CRDs — the same boundary that separates `InboundTrafficPolicy` from `DestinationRule`:

- BackendTrafficPolicy (added v1.3, January 2025): outbound/upstream settings including compression. Targets Gateway, HTTPRoute, GRPCRoute, TCPRoute, UDPRoute, or TLSRoute via targetRefs. The compressor field takes an ordered list of algorithms with per-algorithm tuning and a minContentLength threshold.
- ClientTrafficPolicy: inbound/listener settings including TCP keepalive, connection limits, and HTTP/1–HTTP/2 settings on the client-facing listener. Covers the same gap that `InboundTrafficPolicy.tcp` would address.

kgateway — TrafficPolicy CRD ([docs](#))

kgateway uses a TrafficPolicy CRD that bundles auth, rate limiting, CORS, fault injection, circuit breaking, and compression under one resource. The compression piece lives at `spec.compression` and explicitly separates response compression from request decompression.

Contour — global config file ([docs](#))

A single `compression.algorithm` field in Contour's static config file. Supports gzip, brotli, zstd, or disabled. Applied globally to all listeners; no per-route or per-listener control. Simpler but not comparable in scope to Istio's requirements.

Traefik — Middleware CRD ([docs](#))

Traefik exposes compression via a Middleware resource referenced by name from an IngressRoute rule. Structurally similar to ExtensionRef — a separate resource holding the config, referenced inline from the routing resource. Per-route precision is native since each route chooses which middlewares to apply. `compress: {}` is valid with all defaults; explicit fields are `encodings` (priority-ordered list, default: gzip/br/zstd), `minResponseBodyBytes` (default: 1024), `includedContentTypes/excludedContentTypes`, and `defaultEncoding` (fallback when `Accept-Encoding: *`).

nginx Ingress Controller — ConfigMap ([docs](#))

Compression is configured entirely via a global ConfigMap — there is no per-route or per-Ingress control. Both gzip and brotli are supported. Key fields: `use-gzip` (default: false), `gzip-level` (default: 1), `gzip-min-length` (default: 256 bytes), `gzip-types`; `enable-brotli` (default: false), `brotli-level` (default: 4), `brotli-min-length` (default: 20 bytes), `brotli-types`.

Summary

Implementation	Approach	Granularity
Envoy Gateway	Two CRDs (BackendTrafficPolicy for compression, ClientTrafficPolicy for TCP inbound), policy attachment	Gateway, HTTPRoute, or other route kinds
kgateway	Single CRD (TrafficPolicy), policy attachment	HTTPRoute
Traefik	Dedicated CRD (Middleware), referenced inline from route	Per-route rule
Contour	Global config file	Global only
nginx Ingress	Global ConfigMap	Global only

HBONE compression