Envoy Gateway v1.7: Global Rate Limiting with a Custom ratelimit Service

2026-04-19

How to wire envoyproxy/ratelimit as a self-hosted service into Envoy Gateway v1.7 using EnvoyPatchPolicy — three xDS patches, namespace-admin self-service, and the pitfalls to avoid.

Global rate limiting in Envoy Gateway (EG) can be set up in two ways: the easy path (EG manages its own `envoy-ratelimit` container) and the flexible path (you bring your own envoyproxy/ratelimit service). This post covers the flexible path — **Option B** — and documents the three xDS patches required to wire it up, the EG v1.7 breaking changes that affect the approach, and a namespace-admin self-service deployment model.

Why Rate Limiting in Production

Every public-facing API eventually attracts traffic it wasn't designed for: scrapers that ignore backoff hints, misconfigured clients retrying in tight loops, credential-stuffing bots working through a list, or just a single user whose code went wrong and fires 10 000 requests in a minute. Without a rate limit, all of that hits your backend — and the backend has no way to distinguish it from legitimate traffic until it's already under load.

Global rate limiting addresses this at the proxy layer, before the request reaches any application code:

- **Protect backend capacity.** A sudden spike — intentional or not — is absorbed at the edge. Backends stay healthy and continue serving legitimate traffic.
- **Prevent resource exhaustion.** Databases, caches, and downstream services have finite capacity. A single misbehaving client can exhaust connection pools and degrade the service for everyone else. Rate limiting puts a hard ceiling on per-client throughput.
- **Reduce attack surface.** Brute-force login attempts, credential stuffing, and enumeration attacks all depend on being able to fire many requests quickly. Even a modest limit of 60 requests per minute per IP makes these attacks impractical without being noticeable to normal users.
- **Cost control.** In cloud environments, compute and egress cost money. An unthrottled client that hammers your API can run up a surprising bill before anyone notices.
- **Fairness.** Shared APIs serve many tenants. Without limits, one heavy user degrades latency for all others. Rate limits enforce fairness without requiring explicit per-tenant quotas.

The key property of *global* rate limiting — as opposed to *local* (per-instance) limits enforced independently in each Envoy replica — is that the counter is shared across all replicas. A client that spreads requests across three Envoy pods still hits the same limit. This requires a central counter store (Redis or Valkey) and a dedicated rate-limit service.

One naming caveat: "global" in the Envoy/EG sense means *shared counter*, not *cluster-wide*. In the namespace-admin model described in this post, each namespace runs its own ratelimit service. The limit is global within that service's scope — it is not shared across Gateways in other namespaces. Think of it as *global within the service*, not *global across the cluster*.

Valkey key isolation: ratelimit constructs keys as `{domain}_{descriptor_key}_{value}` (e.g. `envoy-gateway_client_ip_1.2.3.4`). If multiple namespaces share a single Valkey instance and all use `domain: envoy-gateway`, their counters **overlap** — a client rate-limited in namespace A is also counted against namespace B's limit. To isolate counters per namespace, set a unique domain in both the ratelimit ConfigMap and the matching `domain:` field in EnvoyPatchPolicy Patch 2:

```
# ConfigMap config.yaml
domain: ratelimit-workload-test   # unique per namespace

# EnvoyPatchPolicy Patch 2 (must match)
domain: ratelimit-workload-test
```

With unique domains, the Valkey keys are namespaced and do not interfere with each other.

Option A vs Option B

Envoy Gateway supports two integration modes for global rate limiting:

Option A — EG-managed: Set `spec.rateLimit.backend.redis.url` in the `EnvoyGateway` CR. EG deploys and manages its own `envoy-ratelimit` container in `envoy-gateway-system`. Rate-limit rules are expressed via `BackendTrafficPolicy` targeting an `HTTPRoute` — EG translates those rules into xDS automatically. Example:

```
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: ratelimit-policy
  namespace: workload-test
spec:
  targetRef:
    group: gateway.networking.k8s.io
    kind: HTTPRoute
    name: http-echo
  rateLimit:
    type: Global
    global:
      rules:
        - clientSelectors:
            - sourceCIDR:
                value: "0.0.0.0/0"
                type: Distinct   # one counter per distinct source IP
          limit:
```

```
            requests: 5
            unit: Minute
```

`sourceCIDR: 0.0.0.0/0` with `type: Distinct` makes EG generate a `remote_address` descriptor per source IP — EG handles the xDS wiring internally, no manual patches needed. Simple, but you share a single rate-limit service across all namespaces and lose control over its configuration, scaling, and observability.

Option B — Custom deployment: You deploy `envoyproxy/ratelimit` yourself, in any namespace. `EnvoyPatchPolicy` wires Envoy to your service via xDS patches. `BackendTrafficPolicy` is not used — the two approaches are mutually exclusive. Applying `BackendTrafficPolicy.rateLimit.global` without a configured EG ratelimit backend returns HTTP 500 for *all* requests through that HTTPRoute.

Option B is the right choice when: - Namespace admins should own their rate-limit configuration independently - You need per-namespace isolation of rate-limit state - You want to control the ratelimit service lifecycle (image version, replicas, HPA) - You need Prometheus metrics from the ratelimit service itself

Architecture

{{ image(src="/img/envoy-gateway-ratelimit-architecture.drawio.png", alt="Envoy Gateway Global Rate Limiting Architecture") }}

The full stack:

- **Envoy Proxy** (EG-managed, `envoy-gateway-system`) — receives all traffic, runs the HTTP filter chain
- **ratelimit** (`workload-test`) — `envoyproxy/ratelimit:master`, gRPC on `:8081`, metrics on `:9090`
- **Valkey** (`valkey-system`) — 1 Primary + 2 Replicas, ACL auth, stores per-IP counters
- **http-echo** (`workload-test`) — upstream backend, never sees rate-limited requests

Rate limiting is enforced **at the proxy**, before the request reaches the backend. A 429 response has `upstream_host: null` in the Envoy access log — the backend is never contacted.

The Three Required xDS Patches

`EnvoyPatchPolicy` is a Kubernetes-native way to inject raw xDS config into EG-managed Envoy proxies. For Option B, three patches are required:

Patch 1 — CDS Cluster

Adds a new upstream cluster so Envoy can reach the ratelimit gRPC service:

```
- type: "type.googleapis.com/envoy.config.cluster.v3.Cluster"
  name: ratelimit_cluster
  operation:
    op: add
    path: ""
```

```
    value:
      name: ratelimit_cluster
      type: STRICT_DNS
      connect_timeout: "10s"
      http2_protocol_options: {}
      load_assignment:
        cluster_name: ratelimit_cluster
        endpoints:
          - lb_endpoints:
              - endpoint:
                  address:
                    socket_address:
                      address: ratelimit.workload-test.svc.cluster.local
                      port_value: 8081
```

`http2_protocol_options: {}` is required — the ratelimit gRPC API is HTTP/2 only.

Patch 2 — Listener HTTP Filter

Inserts the ratelimit HTTP filter into the Envoy listener's filter chain, **before** the router:

```
- type: "type.googleapis.com/envoy.config.listener.v3.Listener"
  name: "workload-test/eg/http"
  operation:
    op: add
    path: "/default_filter_chain/filters/0/typed_config/http_filters/0"
    value:
      name: envoy.filters.http.ratelimit
      typed_config:
        "@type": "type.googleapis.com/
envoy.extensions.filters.http.ratelimit.v3.RateLimit"
        domain: envoy-gateway
        failure_mode_deny: false
        timeout: "0.25s"
        rate_limit_service:
          transport_api_version: V3
          grpc_service:
            envoy_grpc:
              cluster_name: ratelimit_cluster
```

`failure_mode_deny: false` means fail-open — if the ratelimit service is unreachable, requests pass through. Set to `true` for strict enforcement.

`domain` must match the `domain:` field in the ratelimit ConfigMap.

What `domain` is — and what it isn't: `domain` is not an HTTP hostname or DNS name. It is a plain string that the ratelimit service uses as a key namespace prefix in Redis/Valkey. Every counter the service stores is keyed as `{domain}_{descriptor_key}_{value}` — for example `workload-test_client_ip_1.2.3.4`. It serves two purposes: it selects which block of descriptors in the

config file applies to this request (a single ratelimit service can serve multiple domains from one config), and it isolates counters between deployments that share the same Redis instance. Two namespaces that each set a unique domain value (e.g. `workload-test` and `workload-prod`) will never share a counter even on the same Valkey.

Patch 3 — RouteConfiguration rate_limits

Without this patch, Envoy sends requests to the ratelimit service with *no descriptors* — it has no idea what to ask about. This patch tells Envoy which descriptor to generate per request.

Behind a reverse proxy or load balancer the real client IP arrives in `X-Forwarded-For` (de-facto standard) or `Forwarded` (RFC 7239). Using `request_headers` picks up these headers instead of the TCP source address, which would just be the proxy IP.

Two rules cover both headers. Each rule uses `skip_if_absent: true` so the rule is silently dropped if the header is absent rather than blocking the request:

```
- type: "type.googleapis.com/envoy.config.route.v3.RouteConfiguration"
  name: "workload-test/eg/http"
  operation:
    op: add
    path: "/virtual_hosts/0/rate_limits"
    value:
      # Rule 1: rate-limit by X-Forwarded-For (de-facto standard)
      - actions:
          - request_headers:
              header_name: x-forwarded-for
              descriptor_key: client_ip
              skip_if_absent: true
      # Rule 2: rate-limit by Forwarded (RFC 7239)
      - actions:
          - request_headers:
              header_name: forwarded
              descriptor_key: client_ip
              skip_if_absent: true
```

Both rules write to the same descriptor key `client_ip`. The ratelimit service matches it:

```
domain: workload-test
descriptors:
  - key: client_ip
    rate_limit:
      requests_per_unit: 100
      unit: MINUTE
```

Note: if a request carries *both* headers, it matches both rules and is counted twice per request. In practice, a well-configured upstream proxy sets exactly one of them. If you need strict single-count behaviour, pick one header and remove the other rule.

For environments where neither header is set (direct access, no proxy), add a third fallback rule using `remote_address`:

```
      # Rule 3: fallback to TCP source IP when no proxy headers are
present
      - actions:
          - remote_address: {}
```

Unlike `request_headers`, the `remote_address` action has no `descriptor_key` parameter — it always generates a descriptor with the fixed key `remote_address`. The ratelimit config must therefore list both keys:

```
descriptors:
  - key: client_ip      # matches rules 1 and 2 (header-based)
    rate_limit:
      requests_per_unit: 100
      unit: MINUTE
  - key: remote_address # matches rule 3 (direct access fallback)
    rate_limit:
      requests_per_unit: 100
      unit: MINUTE
```

If `remote_address` is omitted from the config, rule 3 fires but matches nothing — direct-access requests are not rate-limited.

EG v1.7 Breaking Changes

1. EnvoyPatchPolicy must be explicitly enabled

EG v1.7 ships with `EnvoyPatchPolicy` disabled by default. Without this flag, the policy stays in `ACCEPTED: False` with reason `Disabled`:

```
# values.yaml (Helm) or envoy-gateway-config ConfigMap
config:
  envoyGateway:
    extensionApis:
      enableEnvoyPatchPolicy: true
```

Or patch the running ConfigMap directly:

```
kubectl patch configmap envoy-gateway-config \
  -n envoy-gateway-system --type=merge \
  -p '{"data":{"envoy-gateway.yaml":"...\nextensionApis:\n
enableEnvoyPatchPolicy: true\n"}}'
kubectl rollout restart deployment/envoy-gateway -n envoy-gateway-system
```

2. Listener structure: `default_filter_chain`

The Listener xDS resource uses `default_filter_chain` as the main filter chain. Patching the wrong path silently creates a broken entry (missing `@type`) and the policy ends up `PROGRAMMED: False` with an unmarshal error. The correct path is:

```
/default_filter_chain/filters/0/typed_config/http_filters/0
```

ratelimit Container: Distroless Image

`envoyproxy/ratelimit:master` uses a distroless base image — no shell, no package manager. The OCI manifest has no `ENTRYPOINT` or `CMD`, so the deployment must specify it explicitly:

```yaml
containers:
  - name: ratelimit
    image: envoyproxy/ratelimit:master
    command: ["/bin/ratelimit"]
```

Without `command`, the pod crashes immediately with `no command specified`.

The same distroless constraint means no `kubectl exec` debugging — use port-forward to reach the debug endpoint:

```
kubectl port-forward -n workload-test deploy/ratelimit 6070:6070
# check loaded config
curl http://localhost:6070/rlconfig
# envoy-gateway.client_ip: unit=MINUTE requests_per_unit=5
```

ConfigMap Volume: subPath Fix

Mounting the ratelimit ConfigMap without `subPath` causes a `duplicate domain` panic at startup. The `goruntime` library follows Kubernetes' `..data` symlink and loads `config.yaml` twice:

```yaml
# broken — goruntime sees config.yaml twice via ..data symlink
volumeMounts:
  - name: config
    mountPath: /srv/runtime_data/current/config

# fixed — single file, no symlink traversal
volumeMounts:
  - name: config
    mountPath: /srv/runtime_data/current/config/config.yaml
    subPath: config.yaml
    readOnly: true
```

Namespace Admin Self-Service

The entire Option B setup lives in the workload namespace. A namespace admin (who has no cluster-admin access) can deploy rate limiting independently:

```
workload-test/
├── Deployment/ratelimit            # the ratelimit service
├── Service/ratelimit               # :8081 gRPC, :8080 HTTP, :9090
metrics
├── ConfigMap/ratelimit-config      # domain + descriptors (what to
limit)
├── Secret/ratelimit-redis          # Valkey credentials
├── EnvoyPatchPolicy/ratelimit-patch # wires EG to this ratelimit
instance
├── Gateway/eg                      # Gateway API entry point
├── HTTPRoute/http-echo             # route to backend
└── Deployment/http-echo            # the actual workload
```

The cluster admin provides: 1. The Envoy Gateway installation (with `enableEnvoyPatchPolicy: true`) 2. A shared Valkey cluster in `valkey-system`

Everything else is namespace-scoped and namespace-admin-controlled.

Deployment

The full setup — Kustomize base, overlays for AKS, GKE, OpenShift, and local k3s, Helm chart with platform-specific values files, Terraform + Ansible for the test cluster, and the Valkey + Envoy Gateway install scripts — is available at envoy-gateway-ratelimit.

The Kustomize overlays handle platform differences (topology spread constraints on AKS/GKE, OpenShift SCC constraints, namespace-scoped secrets) while sharing a common base. The Helm chart exposes the same knobs via `values.yaml` for teams that prefer Helm.

Kustomize (local k3s overlay)

```
# 1. Create the secret with Valkey credentials
kubectl create secret generic ratelimit-redis \
  -n workload-test \
  --from-literal=REDIS_URL=valkey.valkey-system.svc.cluster.local:6379 \
  --from-literal=REDIS_AUTH=<password>

# 2. Deploy ratelimit into workload-test
kustomize build deploy/kustomize/overlays/local | kubectl apply -f -

# 3. Deploy the test workload (http-echo, Gateway, HTTPRoute,
EnvoyPatchPolicy)
VALKEY_PASSWORD=<password> ./setup/test-workload/install.sh
```

The `overlays/local` kustomization sets `namespace: workload-test` and removes the `ServiceMonitor` (no prometheus-operator on k3s) and placeholder Secret.

Helm

```
helm install ratelimit deploy/helm/ratelimit \
  -f deploy/helm/ratelimit/values-aks.yaml \
  --set redis.secretName=ratelimit-redis \
  -n workload-test --create-namespace
```

Platform-specific values files are provided for AKS, GKE, and OpenShift.

Test

After deployment, verify the ratelimit service loaded its config:

```
kubectl port-forward -n workload-test deploy/ratelimit 6070:6070
curl -s http://localhost:6070/rlconfig
# envoy-gateway.client_ip: unit=MINUTE requests_per_unit=5
```

Check the EnvoyPatchPolicy was accepted and programmed:

```
kubectl get envoypatchpolicy -n workload-test
# NAME              ACCEPTED   PROGRAMMED   AGE
# ratelimit-patch   True       True         2m
```

Fire 8 requests against the gateway (limit is 5/min per source IP):

```
GATEWAY_IP=$(kubectl get svc -n envoy-gateway-system \
  -l 'gateway.envoyproxy.io/owning-gateway-namespace=workload-test' \
  -o jsonpath='{.items[0].status.loadBalancer.ingress[0].ip}')

for i in $(seq 1 8); do
  echo -n "req $i: "
  curl -so /dev/null -w '%{http_code}' "http://${GATEWAY_IP}/"
  echo
done
```

Expected output:

```
req 1: 200
req 2: 200
req 3: 200
req 4: 200
req 5: 200
req 6: 429
req 7: 429
req 8: 429
```

The Envoy access log confirms rate limiting happens before the upstream:

```
{
  "response_code": 429,
```

```
  "response_code_details": "request_rate_limited",
  "response_flags": "RL",
  "upstream_host": null
}
```

What About TCPRoute and TLSRoute?

The HTTP filter chain approach only works for `HTTPRoute` (and `GRPCRoute`, which shares the HCM). For `TCPRoute` and `TLSRoute`, there is no HTTP filter layer — Envoy operates at L4. The `envoy.filters.network.ratelimit` (L4) filter exists but only supports `remote_address` and `destination_port` descriptors. SNI-based rate limiting on `TLSRoute` is not possible without TLS termination, since the payload is encrypted and SNI is only visible in the ClientHello.

What About Istio?

If your cluster runs Istio instead of Envoy Gateway, the same `envoyproxy/ratelimit` service and the same Redis/Valkey backend work — the concepts are identical because Istio also uses Envoy as its data plane. The difference is only in how you wire the integration: instead of `EnvoyPatchPolicy`, Istio uses `EnvoyFilter` to inject the ratelimit HTTP filter and the CDS cluster into the sidecar or gateway proxy.

The ratelimit config (`domain`, `descriptors`) and the descriptor generation logic (header-based or `remote_address`) stay exactly the same. A ratelimit service deployed for an EG setup can be reused in an Istio setup without any changes to the service itself — only the wiring resource changes.

Istio's own documentation covers the `EnvoyFilter`-based approach: Istio — Rate Limiting. The main gotcha when coming from EG: Istio's `EnvoyFilter` targets individual workloads or gateways via `workloadSelector`, and the filter insertion path in the xDS structure differs from EG's `default_filter_chain` layout.

Repository

The full Kustomize base + overlays (AKS, GKE, OpenShift, local), Helm chart, Terraform/Ansible cluster setup, and test workload scripts are in envoy-gateway-ratelimit.