Design, Security Analysis, and Evaluation of Endpoint-Aware Token-Bucket Rate Limiting for Web APIs Using Database-Configured Policies

Kawshik Kumar Paul *

Posted Date: 27 February 2026

doi: 10.20944/preprints202602.1689.v1

Keywords: API rate limiting; token bucket; endpoint-aware rate limiting; endpoint normalization; policy precedence; identity resolution; trusted proxy; client IP extraction; bounded state; adversarial testing

 doi:10.20944/preprints202602.1689.v1

Article

Design, Security Analysis, and Evaluation of Endpoint-Aware Token-Bucket Rate Limiting for Web APIs Using Database-Configured Policies

Kawshik Kumar Paul

Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Bangladesh; kawshikbuet17@gmail.com

Abstract

A common recommendation for preventing brute-force authentication, credential stuffing, scraping, and resource exhaustion is rate limiting, a standard control for API availability and security. Uncertain semantics under horizontal scaling, proxy-induced client-IP ambiguity, unsafe identity binding, ambiguous policy resolution, canonicalization gaps in endpoint matching, and unbounded in-memory state are some of the reasons why deployments frequently fail. In this paper, we propose an endpoint-aware mechanism that uses token buckets with bounded bucket state to enforce per-endpoint RPS limits that are configured in a database and cached for low-latency lookup. The mechanism offers robust endpoint normalization (including an explicit `UNKNOWN` bucket for unmatched paths), deterministic policy precedence, strict identity validation, trusted-proxy boundaries for client IP extraction, optional per-endpoint cost weights, and operational controls for safe defaults and cache reload. We propose adversarial test suites and microbenchmarks for reproducible evaluation and structure a security analysis by attack surfaces (identity, endpoint normalization, state management, configuration, and distributed enforcement). Additionally, we compare the mechanism to commonly used gateway/edge systems and shared-state primitives, and place it within related work on distributed algorithms, adaptive controllers, overload control, and API rate-limit adoption patterns.

Keywords: API rate limiting; token bucket; endpoint-aware rate limiting; endpoint normalization; policy precedence; identity resolution; trusted proxy; client IP extraction; bounded state; adversarial testing

---

1. Introduction

Automated abuse of modern web APIs is constant and includes volumetric floods meant to reduce availability, enumeration and scraping against business endpoints, and brute-force and credential stuffing against authentication endpoints. As stated clearly in API security guidelines and authentication recommendations, rate limiting is thus used as a security control as well as an availability mechanism [1–4].

In practice, many implementations exhibit security-relevant failure modes:

- **Unsafe identity binding**: keying strict limits on attacker-controlled identifiers (e.g., unauthenticated usernames) enables targeted lockouts and cross-user interference.
- **Proxy ambiguity**: incorrect client IP extraction behind reverse proxies causes false positives; trusting forwarding headers from untrusted peers enables spoofing.
- **Endpoint canonicalization gaps**: multiple equivalent URL representations can bypass endpoint-specific limits.
- **State explosion**: per-key bucket creation without bounds enables memory/CPU denial-of-service.
- **Scaling semantics drift**: local-only rate limiters enforce per-replica budgets, which can be exceeded by traffic sprayed across replicas.

 doi:10.20944/preprints202602.1689.v1

This paper proposes a language- and framework-agnostic mechanism that tackles these problems while maintaining tractability: security-aware identity resolution, bounded token-bucket state, fast policy caching, robust endpoint normalization with explicit handling for unknown paths, and per-endpoint RPS limits stored in a database (with deterministic precedence).

Contributions.

- An **endpoint-aware, DB-configured** token-bucket mechanism with deterministic policy precedence and low-latency caching.
- A **security analysis** organized by concrete bypass/DoS vectors: identity binding, proxy/IP trust, canonicalization and endpoint matching, bucket explosion, misconfiguration, and scaling semantics.
- A **reproducible evaluation methodology** (microbenchmarks + adversarial tests) that measures overhead and attacker-induced degradation, including fake-endpoint flooding.
- Positioning against **production systems** and **recent research** on overload control and distributed rate limiting [5–12].

2. Background

2.1. Rate Limiting and HTTP Signaling

When throttling, the server may include `Retry-After` in addition to returning `429 Too Many Requests` [13]. Interoperability is improved and retry storms are decreased with standardized feedback [14]. To promote quota policies and current limits, the IETF HTTPAPI working group specifies the `RateLimit` and `RateLimit-Policy` response headers (standards-track Internet-Draft) [15].

2.2. Algorithm Families

Token buckets are a popular option for API traffic because they regulate the average request rate while permitting bursts up to a configured capacity [16,17]. Rolling-window methods, leaky buckets, and fixed-window counters are alternative families. Redis-cell, a shared-state primitive, implements GCRA and offers rolling-window-like behavior without the need for background refill [18–20].

3. Threat Model and Design Goals

3.1. Threat Model

An attacker can send arbitrary HTTP requests and vary:

- paths and encodings, query strings, and method selection,
- headers (including spoofed forwarding headers unless constrained),
- request bodies (size, nesting depth, values),
- source IPs (single host, distributed botnet, or proxy rotation),
- credential state (no creds / stolen API key / stolen bearer token).

3.2. Security and Correctness Goals

We target five goals:

1. **Bypass resistance**: trivial transformations should not exceed intended limits.
2. **Fairness/isolation**: one principal should not drain capacity intended for others.
3. **No secret leakage**: rate-limit keys and telemetry must not expose sensitive identifiers.
4. **DoS robustness**: attacker traffic should not turn limiter state into memory/CPU exhaustion.
5. **Explicit scaling semantics**: per-replica vs global budgets must be well-defined and testable.

4. Mechanism Overview

The request pipeline is:

 doi:10.20944/preprints202602.1689.v1

`Request` → `Early Validity Check` → `Endpoint Normalize` → `Identity Resolve` → `Policy Resolve (DB-cache)` → `Bucket Key` → `TryConsume` → `Allow/Reject`

The mechanism can run at an application boundary (middleware/filter/interceptor), at an internal service boundary, or behind an API gateway. Boundary enforcement can incorporate verified authentication context (e.g., bearer-token subject, tenant/project id) while keeping strict safety constraints on any payload-derived data.

5. Endpoint Model and Normalization

Endpoint-specific enforcement requires mapping raw requests to stable endpoint templates. Two complementary controls are used.

5.1. Endpoint Validity Registry (Early Rejection)

A registry of *recognized* endpoints can be derived from one of:

- the service routing table (router metadata),
- an OpenAPI specification,
- a curated configuration list for critical endpoints.

Requests that do not match any recognized endpoint can be rejected early (e.g., 404) *before* rate-limit work and authentication parsing, avoiding resource waste under fake-endpoint floods.

5.2. Normalization to Templates with `UNKNOWN` Collapse

Normalization maps a request (m, uri) to a template $m{:}\hat{p}$:

- collapse dynamic path segments to $*$ (or equivalent),
- bind method into the template,
- exclude query strings from the route identity unless explicitly required,
- apply a strict canonicalizer to avoid bypasses (dot segments, repeated slashes, percent-decoding policy).

If no pattern matches, the mechanism returns `UNKNOWN` rather than using the raw URI. This prevents bypass via generating many distinct invalid paths (each producing a unique bucket key).

Listing 1. Endpoint normalization with `UNKNOWN` collapse (pseudocode).

```
def normalize_endpoint(method, raw_path):
    path = canonicalize_path(raw_path) # remove ./, resolve ../, collapse //, decide decode
        policy
    for (pattern, template) in ORDERED_PATTERNS: # first match wins
        if match(pattern, path):
            return template.format(method=method)
    return "UNKNOWN"
```

6. Policy Model and Precedence

6.1. Database Schema

A minimal schema stores per-endpoint RPS limits, optionally scoped to a tenant/project:

Column	Type	Meaning
`endpoint`	string	normalized endpoint template (method+path)
`project_id`	nullable string/int	tenant override; NULL = global
`rps_limit`	int	allowed requests per second for this scope

Two reserved templates are recommended:

- `default`: fallback for recognized endpoints without a specific entry.

 doi:10.20944/preprints202602.1689.v1

- `UNKNOWN`: fallback for requests that do not match any recognized endpoint template.

6.2. Deterministic Policy Precedence

Policies are resolved via a deterministic chain:

$$(m{:}\hat{p},\, t) \to (m{:}\hat{p},\, \varnothing) \to (\texttt{default},\, t) \to (\texttt{default},\, \varnothing) \to (\texttt{UNKNOWN},\, \varnothing)$$

where m is HTTP method, $\hat{p}$ is normalized endpoint template, and t is tenant/project id. The `UNKNOWN` fallback is used when endpoint normalization yields `UNKNOWN` (Section 5).

7. Identity Resolution and Key Construction

7.1. Identity Sources and Validation

Identity resolution is tiered and *validated*:

1. **Verified bearer-token subject** (signature verified, not expired) and derived tenant/project id.
2. **Validated API key id** (key exists and is active; never trust an arbitrary header string).
3. **Fallback to client network identity** (IP or prefix), using trusted-proxy extraction rules.

Avoid attacker-controlled identifiers.

Unauthenticated usernames/emails from request parameters or bodies should not be used as strict primary keys. If a user identifier must be incorporated for login flows, use a dual-control design (Section 9.4) and ensure parsing is constrained (Section 9.3).

7.2. Client IP Extraction Behind Proxies

Forwarding headers (e.g., `Forwarded` [21] or `X-Forwarded-For`) are honored only when the immediate peer is a configured trusted proxy. Otherwise, forwarding headers are ignored and the direct peer address is used.

7.3. Bucket Key Format and Secrecy

The bucket key binds endpoint template, identity, and IP (or IP prefix):

$$k = \texttt{endpoint} \,||\, \texttt{principal} \,||\, \texttt{ip}$$

Sensitive identifiers (API keys, tokens) must never appear in raw form in keys, logs, or metrics. Use a stable internal id or a keyed hash (e.g., HMAC) for non-reversible bucketing.

8. Token Bucket Parameters from RPS

8.1. RPS-to-Bucket Mapping

The database stores a single value: `rps_limit`. Bucket parameters are derived deterministically:

$$\text{refill} = \texttt{rps\_limit tokens per second}, \qquad \text{capacity} = \beta \cdot \texttt{rps\_limit}$$

where $\beta \geq 1$ is a configurable burst factor. Requests consume $w(\texttt{endpoint})$ tokens, where w is an optional per-endpoint weight map.

8.2. Per-Endpoint Consume Weights

Some endpoints are substantially more expensive (e.g., purchase/provisioning, report generation). The mechanism optionally assigns weights:

$$\text{cost per request} = w(\texttt{endpoint}) \in \{1, 2, \dots\}$$

 doi:10.20944/preprints202602.1689.v1

so that the effective request rate is approximately `rps_limit`/w. Weights are kept in code/config (not the database) to keep DB operations simple: DB stores RPS; code maps cost.

9. State Management and Operational Controls

9.1. Policy Cache

To avoid DB queries on the hot path, policy entries are cached in memory:

- key: `(endpoint|project_id)`
- value: `rps_limit`

Cache refresh supports:

- periodic refresh with bounded staleness, and/or
- admin-triggered reload for rapid changes.

9.2. Bucket Cache and Boundedness

Bucket state must be bounded to prevent memory DoS. The bucket cache uses:

- **idle expiration** (TTL after last access),
- **maximum size** (LRU/LFU eviction),
- **optional admission control** (refuse new keys under extreme churn).

Eviction implies that a returning principal may receive a full bucket after a long idle period, which is typically acceptable because sustained rate remains bounded by refill.

9.3. Body Parsing Safety

If any endpoint requires extracting fields from request bodies for rate limiting (e.g., login identifier), strict safety constraints are required:

- parse only on an explicit allowlist of endpoints,
- enforce maximum body size and maximum JSON depth,
- fail safely: parsing errors must not silently skip strict checks for sensitive endpoints.

9.4. Dual-Bucket Controls for Sensitive Unauthenticated Flows

For authentication endpoints, a dual-control approach reduces both volumetric abuse and targeted lockouts:

- per-IP (or IP-prefix) bucket to bound volumetric traffic,
- per-identifier bucket to slow targeted guessing,

and require both to pass [1,3].

10. Scaling Semantics and Distributed Enforcement

Local in-memory buckets enforce *per-replica* limits. If a global budget is required, the mechanism can be paired with:

- a global rate-limit service used by gateways/service meshes (descriptor-based enforcement) [22,23],
- shared-state primitives (e.g., Redis/GCRA via redis-cell compatible commands) [18,20],
- distributed algorithms and control planes for large-scale settings [7,8,11,12].

A hybrid approach is often practical: local limiter for low latency and bulk protection, plus a shared/-global limiter for strict endpoints.

11. Security Analysis

We organize the analysis by pipeline stages.

 doi:10.20944/preprints202602.1689.v1

11.1. Identity Binding Failures

Attacker-controlled identifiers.

Using unauthenticated identifiers (e.g., usernames in query/body) as primary strict keys enables targeted lockouts and cross-user interference. Use verified credentials, or dual-bucket controls for authentication endpoints (Section 9.4).

Fake API key rotation.

If arbitrary header strings are accepted as API keys without validation, attackers can generate unbounded unique identities and evade per-key limits. Validation must precede identity selection; invalid keys must fall back to IP-based limiting.

Secret leakage via telemetry.

Keys must not contain raw secrets; metrics tags must avoid high-cardinality identifiers and must not include raw tokens/API keys. Prefer stable ids or keyed hashes.

11.2. Proxy Ambiguity and Header Spoofing

Trust forwarding headers only from configured trusted proxies. Otherwise, ignore them and use the direct peer address [21].

11.3. Endpoint Canonicalization Bypasses

Attackers exploit equivalent representations:

- repeated slashes (`//`), dot segments (`/./`, `/../`),
- percent encoding and double-encoding,
- trailing slash variation, case variation (depending on routing rules),
- query-string abuse when it is included in keys.

Mitigate with a strict canonicalizer and method-bound templates (Section 5). Query strings should not affect endpoint identity unless explicitly required.

11.4. Bucket Explosion and Fake-Endpoint Flooding

Per-principal state can be exploited by generating many unique keys. The bounded bucket cache (Section 9.2) prevents unbounded growth. Additionally:

- early invalid-endpoint rejection reduces wasted work,
- `UNKNOWN` template collapse prevents bypass by random paths,
- strict `UNKNOWN` RPS limits constrain residual work on unmatched-but-not-rejected requests.

11.5. Misconfiguration Hazards

DB-configured policies enlarge the configuration surface: overlapping templates, missing defaults, non-positive limits, and ambiguity in precedence. Mitigate via:

- linting/validation on load (reject non-positive `rps_limit`, detect overlap),
- explicit precedence rules,
- explicit fail-open vs fail-closed behavior for DB/cache outages.

12. Evaluation Methodology

We recommend evaluating both **overhead** and **adversarial robustness**.

12.1. Microbenchmarks

We report per-stage overhead using latency percentiles (P50/P95/P99) to isolate the cost of endpoint normalization, identity resolution, policy lookup, and bucket consumption. A checklist of recommended metrics is provided in Appendix C (Table 2).

Measure:

 doi:10.20944/preprints202602.1689.v1

- endpoint normalization cost (match success, worst-case no match),
- identity resolution cost (verified token, API key validation, IP fallback),
- policy cache lookup and reload overhead,
- bucket consume under contention and under churn,
- end-to-end middleware overhead vs baseline.

12.2. Adversarial Test Suite

A1: Targeted lockout test (auth endpoint).

Compare single-key vs dual-bucket controls (identifier + IP) and measure victim impact and attacker cost [3].

A2: Forwarding header spoofing.

Send spoofed `Forwarded`/`X-Forwarded-For` from untrusted peers and verify it is ignored; verify correct parsing from trusted proxies [21].

A3: Canonicalization corpus.

Construct URI variants and assert they normalize to the same endpoint template and policy.

A4: Bucket explosion under churn.

Generate many unique principals and measure memory, eviction behavior, and latency degradation.

A5: Fake API key rotation.

Send requests with random API-key strings and ensure invalid keys collapse to IP-based limiting.

A6: Fake endpoint flooding.

Flood the system with random paths; verify early rejection and/or `UNKNOWN` collapse prevents bypass via unique-path keying.

A7: Scaling semantics.

Deploy k replicas and measure effective admitted rate under per-replica enforcement; compare against shared-state/global enforcement [18,22].

13. Related Work

Serbout et al.'s study of adoption patterns and documentation practices ([5]) encourages multi-faceted policies (per route, per consumer, per tenant) and clear client communication.

Rate limiting is an entry-point control used in overload control research to maintain SLOs in microservices. Using global observations and rate controllers, TopFull resolves overloads through per-API admission control [6]. Learning-based controllers for microservice environments are also investigated by adaptive approaches [9].

The classic work on coordinated distributed limiters [11] and more recent algorithms and systems [7,8,12] both examine distributed and large-scale enforcement. In addition to server-side enforcement and standardized signaling [13,15], client-side coordination has been proposed to decrease wasted retries and increase shared-quota efficiency [10].

14. Comparison with Production Systems

Table 1 positions the mechanism among common systems and primitives.

 doi:10.20944/preprints202602.1689.v1

Table 1. Comparison with common production systems and primitives.

System	Layer	State	Typical keying	Notes
Endpoint-aware DB policy + token bucket	App/ service boundary	Local (opt. shared)	method+template + principal + IP	Rich identity + endpoint isolation; bounded caches; `UNKNOWN` collapse.
Envoy local/global rate limiting	Gateway/ mesh	Local / external service	route descriptors	Local token bucket and global enforcement via an external rate-limit service [22,24].
Descriptor-based rate limit services (Lyft/Envoy)	Shared service	Shared store	domain + descriptors	Configuration-driven, shared-state decisions returned to callers [23].
NGINX `limit_req`	Edge proxy	shared-memory zone	typically IP-based key	Very fast edge throttling; limited application identity semantics [25].
HAProxy stick tables	Edge/LB	local (+ peers)	IP / arbitrary string keys	Flexible counters; optional synchronization [26].
AWS API Gateway throttling	Managed gateway	provider-managed	account/stage/ route limits	Burst + steady-state model; 429 responses [27].
redis-cell / Dragonfly `CL.THROTTLE` (GCRA)	Datastore primitive	central atomic op	any key	Rolling-window-like behavior, O(1) command [18,20].

15. Discussion and Limitations

Rate limiting is not a complete security solution.

Attackers can remain within budgets while exploiting cost asymmetries or shifting pressure to other layers. Large-scale botnet attacks require edge defenses (CDN/WAF), and connection-exhaustion attacks require server/network-level mitigations [4].

Operational clarity matters as much as algorithms.

The mechanism emphasizes deterministic precedence, bounded state, validated identity selection, and explicit behavior for unknown endpoints. These choices reduce ambiguity-driven security failures.

Scaling semantics must be explicit.

Per-replica enforcement can be acceptable if documented and tested. If global semantics are required, use global services or shared-state primitives.

16. Conclusion

An endpoint-aware, policy-configured token-bucket rate-limiting mechanism for web APIs was introduced in this paper. Per-endpoint RPS limits are stored in a database, deterministic precedence is enforced, endpoints are safely normalized (including `UNKNOWN` collapse), identity sources are verified, and in-memory bucket state is bound to prevent state-explosion DoS. In addition to offering a reproducible evaluation methodology, we offered a security analysis covering identity resolution, proxy trust, canonicalization, state management, misconfiguration, and distributed enforcement. Treating rate limiting as a security mechanism with explicit semantics, bounded resources, and adversarial testing reveals a workable route to reliable deployments when compared to popular gateway/edge systems and current research.

A. Appendix A: Canonicalization Test Corpus

The following inputs should normalize to the same route identity under a chosen canonicalizer:

 doi:10.20944/preprints202602.1689.v1

- `/api/x/../login` → `/api/login`
- `/api//login` → `/api/login`
- `/api/login%2f` (decoding policy must be explicit)

Implementations must define decoding rules to avoid double-decode bypasses.

B. Appendix B: Bucket Cache Requirements

A production bucket cache should implement:

- idle expiration (inactive key eviction),
- size-based eviction (LRU/LFU),
- metrics on bucket creation and eviction,
- optional admission control when churn is high.

C. Appendix C: Evaluation Metrics Checklist

To improve comparability across implementations and workloads, this appendix lists a minimal set of per-stage latency metrics that can be reported as percentiles. Values are expected to be workload- and deployment-dependent.

Table 2. Suggested overhead metrics to report (percentiles; μs).

Operation	P50 (μs)	P95 (μs)	P99 (μs)
Normalize endpoint (match)			
Normalize endpoint (no match)			
Resolve identity (token)			
Resolve identity (API key)			
Resolve identity (IP fallback)			
Policy cache lookup			
Bucket consume (single-thread)			
Bucket consume (contended)			
End-to-end limiter overhead			

References

1. OWASP Foundation. API Security Top 10 (2023): API2 – Broken Authentication. https://owasp.org/API-Security/editions/2023/en/0xa2-broken-authentication/, 2023. Accessed 2026-02-16.
2. OWASP Foundation. API Security Top 10 (2019): API4 – Lack of Resources & Rate Limiting. https://owasp.org/API-Security/editions/2019/en/0xa4-lack-of-resources-and-rate-limiting/, 2019. Accessed 2026-02-16.
3. Temoshok, D.; et al. Digital Identity Guidelines: Authentication and Authenticator Management. Special Publication NIST SP 800-63B, National Institute of Standards and Technology (NIST), 2025. Revision 4. Accessed 2026-02-16.
4. OWASP Cheat Sheet Series. Denial of Service Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Denial_of_Service_Cheat_Sheet.html, 2026. Accessed 2026-02-16.
5. Serbout, S.; El Malki, A.; Pautasso, C.; Zdun, U. API Rate Limit Adoption – A Pattern Collection. In Proceedings of the Proceedings of the 28th European Conference on Pattern Languages of Programs (EuroPLoP '23). Association for Computing Machinery, 2023, pp. 5:1–5:20. https://doi.org/10.1145/3628034.3628039.
6. Park, J.; Park, J.; Jung, Y.; Lim, H.; Yeo, H.; Han, D. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. In Proceedings of the Proceedings of the ACM SIGCOMM 2024 Conference. Association for Computing Machinery, 2024, pp. 876–890. https://doi.org/10.1145/3651890.3672253.
7. Chen, Z.; Fan, Y.; Qian, K.; Meng, Q.; Shu, R.; Li, X.; Zhang, Y.; Wang, B.; Li, W.; Ren, F. ScalaTap: Scalable Outbound Rate Limiting in Public Cloud. In Proceedings of the IEEE INFOCOM 2025, 2025, pp. 1–10. https://doi.org/10.1109/INFOCOM55648.2025.11044509.

 doi:10.20944/preprints202602.1689.v1

8. Chen, L.; et al. CMDRL: A Markovian Distributed Rate Limiting Algorithm in Cloud Networks. In Proceedings of the Proceedings of APNet 2024. Association for Computing Machinery, 2024. https://doi.org/10.1145/3663408.3663417.
9. Lyu, N.; Wang, Y.; Cheng, Z.; Zhang, Q.; Chen, F. Multi-Objective Adaptive Rate Limiting in Microservices Using Deep Reinforcement Learning. https://arxiv.org/abs/2511.03279, 2025. arXiv:2511.03279. Accessed 2026-02-16.
10. Farkiani, B.; Liu, F.; Crowley, P. Rethinking HTTP API Rate Limiting: A Client-Side Approach. https://arxiv.org/abs/2510.04516, 2025. arXiv:2510.04516. Accessed 2026-02-16.
11. Raghavan, B.; Vishwanath, K.; Ramabhadran, S.; Yocum, K.; Snoeren, A.C. Cloud Control with Distributed Rate Limiting. In Proceedings of the Proceedings of the ACM SIGCOMM 2007 Conference. Association for Computing Machinery, 2007, pp. 337–348. https://doi.org/10.1145/1282380.1282419.
12. Guan, B. Designing Scalable Rate Limiting Systems: Algorithms, Architecture, and Distributed Solutions. https://arxiv.org/abs/2602.11741, 2026. arXiv:2602.11741. Accessed 2026-02-16.
13. Nottingham, M.; Fielding, R.T. Additional HTTP Status Codes. RFC 6585, 2012.
14. MDN Web Docs. HTTP 429 Too Many Requests. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/429, 2026. Accessed 2026-02-16.
15. IETF HTTPAPI Working Group. RateLimit Header Fields for HTTP. https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-ratelimit-headers-10, 2025. Internet-Draft draft-ietf-httpapi-ratelimit-headers-10 (Sep 27, 2025). Accessed 2026-02-16.
16. Heinanen, J.; Guerin, R. A Single Rate Three Color Marker. RFC 2697, 1999.
17. Heinanen, J.; Guerin, R. A Two Rate Three Color Marker. RFC 2698, 1999.
18. Leach, B. redis-cell. https://github.com/brandur/redis-cell, 2026. Accessed 2026-02-16.
19. Leach, B. Rate Limiting, Cells, and GCRA. https://brandur.org/rate-limiting, 2015. Accessed 2026-02-16.
20. DragonflyDB. CL.THROTTLE Command Reference. https://www.dragonflydb.io/docs/command-reference/strings/cl.throttle, 2026. Accessed 2026-02-16.
21. Fielding, R.T.; Reschke, J. Forwarded HTTP Extension. RFC 7239, 2014.
22. Envoy Project. Global Rate Limiting Architecture Overview. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_features/global_rate_limiting, 2026. Accessed 2026-02-16.
23. Envoy Proxy Community. ratelimit: A Generic gRPC Rate Limit Service (Envoy-Compatible). https://github.com/envoyproxy/ratelimit, 2026. Accessed 2026-02-16.
24. Envoy Project. HTTP Rate Limit Filter Documentation. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rate_limit_filter, 2026. Accessed 2026-02-16.
25. NGINX. Module ngx_http_limit_req_module. https://nginx.org/en/docs/http/ngx_http_limit_req_module.html, 2026. Accessed 2026-02-16.
26. HAProxy Technologies. Stick Tables Configuration Tutorial. https://www.haproxy.com/documentation/haproxy-configuration-tutorials/proxying-essentials/custom-rules/stick-tables/, 2026. Accessed 2026-02-16.
27. Amazon Web Services. Amazon API Gateway Request Throttling. https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html, 2026. Accessed 2026-02-16.