

Trishula: An eBPF-Native, Developer-First Web Application and API Firewall for Kubernetes

Shain Singh (shain.singh@owasp.org)

trishula-dev project · github.com/trishula-dev · trishula.dev · Melbourne, Australia

Revision 1.0 – 2 October 2026 · Work in progress. Full product specification companion: “Trishula – Product Requirements Document & Project Charter” (v4).

Abstract. The web application firewall is the last major security control that never crossed into developer ubiquity: iptables, mTLS, OAuth/OIDC, and secrets vaults all became pervasive because tooling made them usable by ordinary developers, while the WAF remained an appliance-shaped system of vendor dialects, out-of-band consoles, and tuning engagements – with the practical consequence that most Kubernetes estates run either an awkwardly bolted nginx hop or nothing at all. We present Trishula, an open-source (Apache-2.0), eBPF-accelerated web application and API firewall for Kubernetes that attempts the crossing. Trishula is three deployables compiled from one policy source: a per-node eBPF shield (XDP at the NIC, TC at the interfaces) that stops floods, enforces access-control maps, and drops repeat offenders in roughly a microsecond without consulting userspace; a userspace engine that reconstructs full HTTP/1.1, HTTP/2, and gRPC sessions and evaluates a seven-family detection ladder – OWASP Core Rule Set signatures (with verdict-level parity against an embedded Coraza reference evaluator enforced by a deliberately differential CI gate), a Common Expression Language (CEL) rule surface over a structured request view, OpenAPI positive-security enforcement with API discovery, bot detection across TLS-fingerprint, header-plane, and behavioural families, per-route rate limiting, a specified, versioned ScoreWindowBan temporary-ban algorithm (fail2ban’s operational UX re-specified for HTTP and enforced in kernel maps), and an in-band ML tier with an async semantic-judge escalation path; and a Kubernetes operator that compiles CRD policy into signed bundles, compiled matcher databases, and kernel map values. The engine is a plain Gateway-API citizen – a health-checked, weight-steerable Service backend on any conformant data plane (proved first on NGINX Gateway Fabric), not a proprietary proxy hop. Every verdict is an OpenTelemetry log, span, and metric correlated by trace identifier, and every documented rule cites the wire bytes it inspects in a teach-the-web curriculum shipped as a product deliverable. We argue the position: that positive security, bot defence, specified temporary bans, and kernel altitude are exactly the features the OSS field lacks, that honesty mechanisms (shadow-first promotion, differential parity, benchmark-labelled-as-target-until-measured) are load-bearing rather than promotional, and that developer comprehensibility – not detection depth alone – is the adoption path for web application and API security. Contributions include the two-plane inspection decomposition with a decision-free kernel, the named ban algorithm with conformance-test specification, and a developer-experience charter whose commitments are CI-enforced.

Keywords – web application firewall · API security · eBPF · XDP · Kubernetes · Gateway API · OWASP Core Rule Set · positive security · OpenAPI · bot detection · JA4 · rate limiting · temporary bans · OpenTelemetry · CEL · Kubernetes operator.

- Trishula: An eBPF-Native, Developer-First Web Application and API Firewall for Kubernetes
- 1. Introduction
 - 1.1 The problem: the control that never crossed
 - 1.2 The design thesis in one sentence
 - 1.3 Why now
 - 1.4 Contributions
 - 1.5 Paper structure
- 2. Background and Related Work
 - 2.1 The field, honestly surveyed
 - 2.2 What we borrow, with credit
 - 2.3 The gap in one picture
- 3. Architecture
 - 3.1 The two-plane principle and the request path
 - 3.2 Gateway-API citizenship (the traffic contract)
 - 3.3 The three deployables
 - 3.4 Operating modes and the adoption ladder
- 4. The Detection Stack
 - 4.1 The ladder
 - 4.2 CRS as the correctness baseline – and the parity gate as the honesty mechanism
 - 4.3 Bot detection as evidence accumulation
 - 4.4 Positive security: OpenAPI as policy, discovery as onboarding
 - 4.5 The ban algorithm: ScoredWindowBan v1, specified
 - 4.6 Rate limiting (the limits half of limits-and-bans)
 - 4.7 The ML tier with a clean hand-off contract
- 5. Developer Experience: the Charter, the OTel Narration, the Curriculum
 - 5.1 A charter enforced by CI
 - 5.2 CEL as the rule language (and why not a new DSL)
 - 5.3 Every verdict narrated: OTel as first-class product surface
 - 5.4 Teach the web (the curriculum as a product deliverable)
- 6. Community, Governance, and the Rules Question
 - 6.1 The footprint
 - 6.2 Community-sourced rules, gated honestly
 - 6.3 Governance and supply-chain posture
- 7. Performance: Targets, Stated as Targets
- 8. Threat Model of Trishula Itself
- 9. Limitations and Future Work
- 10. Conclusion
- 11. References

1. Introduction

1.1 The problem: the control that never crossed

Over the past two decades, a succession of security controls crossed from specialist appliance to developer ubiquity. Packet filtering became `iptables` rules in every deployment guide. Transport trust became mTLS, then automatic via

cert-manager and service meshes. Authentication became OAuth/OIDC libraries developers import in an afternoon. Secrets became vaults with Kubernetes-native injection. Every one of these crossed because three things happened at once: the configuration became declarative, the declaration moved into the developer's own tooling and review flow, and the enforcement point moved into the path the developer already owns. The pattern is consistent: **security wins when developers adopt it, and developers adopt what they can read.**

The web application firewall never made the crossing. It remained appliance-shaped: a reverse-proxy tier of its own, a vendor rule language (ModSecurity directives, F5 profiles), a console operated by a specialist team, and a tuning engagement measured in weeks [322]. In the Kubernetes estate specifically, the WAF is either a separate proxy hop on the application path – extra latency, extra TLS fan-out, a second scaling unit, a config surface nobody on the application team owns – or, in the majority of developer-run clusters, **nothing**, because WAFs are that bad to adopt and the alternative (nothing) is free and immediate.

The gap widens precisely when traffic becomes most sensitive. Kubernetes-native HTTP ingress has consolidated around Gateway API, and AI-inference traffic is becoming its dominant workload: long bodies, streaming responses (SSE), token-metered endpoints, multi-tenant model pools. The request shapes are adversarial surface on the same wire as classic OWASP injection: prompt injection, jailbreaks, training-data exfiltration, credential stuffing. The incumbent answers – an out-of-band proxy hop with its own configuration language, or a cloud WAF that ships payloads to third-party inspection – are both structurally hostile to the people whose APIs they protect.

1.2 The design thesis in one sentence

Security wins when developers adopt it; Trishula is a web application and API firewall a developer can read, compiled from Git through Kubernetes CRDs into kernel and userspace enforcement altitudes, with the OpenTelemetry narration of its own decisions as a first-class product surface.

Or, compressed further: *the WAF as a Gateway-API citizen with a kernel bodyguard*. The kernel bodyguard (the eBPF shield) handles what the kernel is good at – drops, floods, caches, bans at line rate; the citizen (the userspace engine) handles what a programmable userspace is good at – full HTTP reconstruction, regex-scale matching, schema semantics, machine learning; and a single CRD surface compiles one policy into both.

1.3 Why now

Three curves intersect in 2026:

1. **The gateway-API consolidation.** HTTP ingress in Kubernetes has standardised on Gateway API with conformant implementations (NGINX Gateway Fabric, Envoy Gateway, kgateway, Istio, Traefik), and the Gateway API Inference Extension ([39], [41], [43]) extends the pattern to model-pool steering. A security layer that is a plain conformant backend can now be attached by every data plane at once – the integration substrate for a WAF exists that did not five years ago.
2. **eBPF's maturity.** XDP drop rates above 10 Mpps/core on commodity hardware are measured in production environments [60], [75], [76]; ring buffers, CO-RE portability, and Go-native loaders (cilium/ebpf) [94], [126] make kernel/userspace split systems maintainable by small teams. The eBPF fast path is no longer a research project; it is an engineering choice.
3. **AI traffic arrives at the WAF's front door.** Inference endpoints carry prompt-injection and exfiltration payloads through the same HTTP ingress as every other application. Detection needs (semantic analysis, response-side exfiltration detection, per-model-pool policy) exceed what regex-era signatures can express, while the payloads themselves (long JSON bodies, streaming responses) stress the throughput assumptions every WAF made.

1.4 Contributions

This paper makes the following contributions:

1. **A positioning argument with feature-level receipts (Section 2):** the OSS web-application-and-API-security field stops at negative signatures; the missing depth (positive security, bot defence, specified temporary bans, kernel altitude, parity discipline) is enumerated as the product, not as an aspiration.
2. **A two-plane inspection architecture with a decision-free kernel (Section 3):** XDP/TC shield for everything cheap (L3/L4 ACLs, flood tarpits, verdict caches, ban enforcement, flow classification, ring-buffer events) and a userspace engine for everything that needs full HTTP, with the explicit rule that the kernel computes no scores and evaluates no rules – it applies verdicts computed elsewhere, at line rate.
3. **A seven-family detection ladder compiled from one policy source (Section 3–4):** CRS with verdict-level differential parity against an embedded reference evaluator as a CI-blocking gate; a cost-budgeted CEL rule surface over a structured request view; OpenAPI 3.1 positive security with API discovery/learn mode; three-family bot detection with consensus merging; per-route rate algorithms; a named, versioned ScoredWindowBan algorithm; and an ML tier with a clean hand-off contract.
4. **A specified temporary-ban algorithm (Section 4, §4.5):** ScoredWindowBan v1 – fail2ban’s findtime/threshold/bantime/recidivism vocabulary [147] re-specified for HTTP, computed on the wire, enforced in kernel LRU maps, shadow-first in promotion, with evidence records (the contributing score trajectory) attached to every ban and a namespaced `BanRelease` path for false-positive victims.
5. **A developer-experience charter enforced by CI (Section 5):** ten measurable experiences – from “first custom rule ships in ≤ 15 minutes from clone” to “every verdict self-explains with a link to the wire-level lesson it teaches” – each with a stated CI check, on the argument that *understandability is the security feature*.
6. **A governance surface for the OSS field (Section 6):** CRS exclusions as versioned, shadow-tested CRs with a cadence discipline; community-sourced rule corpora under a quality gate that keeps the parity guarantee honest; the org and domain footprint (github.com/trishula-dev, trishula.dev) as the split boundary between product code and community content.

1.5 Paper structure

Section 2 grounds the design in the field it sits in (what we borrow, where the gap is). Section 3 presents the architecture: the two planes, the request path, the modes. Section 4 the detection stack: the ladder, each family, the ban algorithm specified in full. Section 5 the developer-experience charter and observability. Section 6 community, governance, and the repository footprint. Section 7 the performance targets framed honestly as targets pending PoC measurement. Section 8 the threat model of Trishula itself. Section 9 limitations and phasing. Section 10 concludes.

2. Background and Related Work

2.1 The field, honestly surveyed

The rule of engagement for every project below (and for the wider field surveyed in the PRD’s Appendix E – eBPF security tools, runtime-enforcement platforms, gateway plugin stacks, the temporal-ban lineage): **composition over forking, borrowing over rebuilding, and a stated reason when neither is possible.**

The WAF-shaped core:

- **Coraza** [134], [133]: Go WAF library, CRS-compatible (SecLang import), WASM plugin surface. Trishula embeds it as the **reference evaluator** – composition, not fork: CRS fidelity is upstream’s to maintain, Trishula’s job is altitudes and lifecycle around it [312].
- **ModSecurity** [316], [167]: the historical engine. EOL on nginx; libModSecurity in maintenance mode; the community’s CRS work has pivoted to Coraza [310]. Its fate is the cautionary tale: engine stagnation kills rule ecosystems.

- **SafeLine** [196], [205]: container WAF, nginx core, staged semantic/LLM pipeline – the cheap-first cascade shape Trishula’s ML tier adopts, minus the docker-compose-first, nginx-coupled shape.
- **open-appsec** [192], [239]: ML WAF with learning→enforcement lifecycle and Kubernetes CRDs. The lifecycle shape (learn → detect → prevent with objective promotion gates) is borrowed directly; the coupling is not – open-appsec attaches enforcement to a central management plane rather than Gateway-API steering, and has no eBPF altitude.
- **CrowdSec** [145], [170]: community blocklists and the bouncer pattern – one optional ban-engine input; but its HTTP altitude is log-tap, not wire.
- **fail2ban** [147]: the temporal ban UX archetype – findtime/maxretry/bantime vocabulary, recidivism – whose operational success is undisputed and whose altitude (log parsing after the fact, filesystem state, no HTTP session view, no per-route policy) is exactly wrong. Trishula moves the idea onto the wire, into the kernel, and into the CRD (Section 4.5).

The eBPF-native cousins (the wider field): **Synapse** (XDP NDR + proxy, JA4+ blocking, Wirefilter WAF) [F-synapse], **Moat** (Rust XDP edge proxy, JA4+ suite) [F-moat], **MicroWAF** (host WAF with CEL rule engine – independent validation of the CEL-for-WAF choice) [F-microwaf], and the Cloudflare XDP lineage (L4Drop, Katran) [61], [76] that proves the kernel budgets. The common gap, stated once: **every one of them stops at network-shaped policy** (IPs, fingerprints, rates, drops). None compiles CRD policy into kernel maps plus userspace evaluator chains as one artifact; none speaks Gateway API; none carries a signature corpus or a positive-security model; none treats shadow-first correctness as a mechanism. That gap is exactly Trishula’s position (Section 4.3 of the PRD).

2.2 What we borrow, with credit

Trishula’s design is a composition with acknowledged ancestry: Coraza as the embedded CRS reference evaluator [134]; Cilium’s eBPF-fast-path-plus-userspace-L7-proxy decomposition [85], [87] as the validated two-plane pattern; open-appsec’s learning lifecycle shape minus the central plane [239]; SafeLine’s cheap-first staged detection [205]; CrowdSec’s community-intel-as-input, CRD-gated [170]; fail2ban’s ban UX vocabulary [147]; Synapse’s JA4+/kernel-altitude posture [F-synapse]; MicroWAF’s CEL authoring surface [F-microwaf]. Where a project owns an altitude better than Trishula ever will – Coraza’s CRS fidelity, Tetragon’s in-kernel syscall enforcement, Cilium’s east-west policy – the design defers to it explicitly. Trishula is not a replacement for the stack below it; it is the missing north-south layer that makes the stack comprehensible to developers.

2.3 The gap in one picture

Table 1 states the field’s coverage of the features that matter. The x-axis is detection depth (regex-only vs. full stack with positive security, bots, rates, ML); the y-axis is shape (host/app appliance vs. Kubernetes-native). No OSS project occupies the top-right quadrant, and no project at all occupies the quadrant with kernel altitude attached.

Table 1: Where Trishula sits in the OSS field (feature coverage)

Capability	Coraza	SafeLine	open-appsec	CrowdSec	eBPF cousins	Trishula
CRS evaluation	✓ ref	✓	signatures	bridge	–	✓ embedded + parity gate
Positive security (OpenAPI)	–	–	✓	–	–	✓ learn + enforce
API discovery/learn mode	–	–	part	–	–	✓
Bot defence (multi-family)	–	heuristic	✓	✓ log-tap	fingerprint	✓ 3-family + consensus
Rate limiting (per-route CRD)	–	✓	✓	✓ host	host	✓ 4 algorithms, CRD-selectable
Temporary bans, specified algorithm	–	✓ (status-heuristic)	–	✓ log-tap	ad-hoc	✓ ScoredWindowBan v1, conformance-tested

Capability	Coraza	SafeLine	open-appsec	CrowdSec	eBPF cousins	Trishula
Kernel-speed enforcement	–	–	–	–	✓ L4-class	✓ ban tables + verdict caches in-kernel
Kubernetes-native CRD → compile	–	–	CRDs, central	–	–	✓ CRDs → maps+bundles, one artifact
Gateway-API steering citizen	–	–	–	–	–	✓ plain backend, any conformant plane
OTel verdict narration (logs+traces+metrics)	–	–	–	–	–	✓ every verdict a correlated span
Teach-the-web curriculum as deliverable	–	–	–	–	–	✓ docs double as the HTTP/TLS/eBPF textbook

The column Trishula seeks is not “best in each row”; it is every row in one compiled artifact, plus the two rows (kernel altitude, Gateway-API citizenship) nobody else attempts.

3. Architecture

3.1 The two-plane principle and the request path

Trishula splits inspection by altitude. The **kernel plane** (shield) is bounded, allocation-free, and verdict-cheap: L3/L4 ACLs (LPM tries), SYN/UDP flood tarpits, verdict caches, ban tables, flow classification, per-flow behavioural counters, and a bounded plaintext fast path on legs where TLS has already terminated (method-by-load-compare, path-prefix gates, malformed-first-line detection) [60], [61], [79]. The **userspace plane** (engine) has unbounded parsing freedom: full HTTP/1.1, HTTP/2, gRPC reconstruction happens exactly once per request, in Go, where loops, allocations, regex-scale matching, schema semantics, and ML are engineering choices rather than verifier gymnastics.

The split has two hard technical reasons and one design reason. The technical reasons: the eBPF verifier's instruction and loop budgets make real L7 parsing a tail-call gymnastics project with miserable maintenance economics [96], [106], [105]; and the ciphertext problem – at the NIC, HTTP payloads are TLS records, so decisive L7 analysis requires post-termination plaintext, which the kernel plane sees only on already-terminated legs [79]. The design reason is auditable enforcement: in-kernel rule evaluation fails the auditability test even where the verifier would permit it; a kernel that computes nothing can be verified to enforce exactly what userspace decided.

The request path, end to end: client → gateway (NGF in the reference deployment: TLS termination, HTTPRoute matching, edge policy) → trishula-engine Service as a plain weighted backendRef; the engine parses, reconstructs, and walks the ladder; verdicts allow/forward (transparent, stream-aware, no endpoint logic), block (local response: per-route body, or 429 + Retry-After for bans), challenge (evidence-of-work via signed redirect token – no third-party CAPTCHA in the default path), or log/shadow. The shield on each node touches the flow twice for free: XDP at the NIC (ACLs, floods, ban-table and cache hits served without a userspace round-trip) and TC at engine-node veths (flow classification, rate counters, ring-buffer events). Ban writes are the ring's always-sampled high-priority class – enforcement state flows kernel-ward, never payload-ward.

One policy artifact compiles to many altitudes: a RuleSet CR compiles simultaneously to (a) userspace evaluator chains, (b) a Vectorscan matcher database for the hot regex subset [128], [172], [173], (c) kernel map values (ACL tries, verdict caches, ban tables), and (d) OTel counters. One source of truth, no drift.

3.2 Gateway-API citizenship (the traffic contract)

The v1 integration contract requires nothing beyond conformant Gateway API. NGINX Gateway Fabric (NGF) [F-ngf27] is the reference data plane – Apache-2.0, hardened h2c upstreams, weighted backendRefs, an active release train – chosen because it is boring in the right places, not because the design depends on it. The contract's terms, each CI-tested in the kind/NGF reference lab (§10.3 of the PRD):

1. **Steering** – the gateway steers to trishula-engine as it would to any Service: weighted backendRefs, header matches, canary weights. Trishula never selects endpoints; engine-owned endpoint selection is structurally absent (the forward pass is deliberately dumb: resolve steered Service endpoints, preserve hop-by-hop semantics, stream with bounded chunk buffering).
2. **Health** – engine readiness equals current-policy-bundle load state (not process liveness); unhealthy replicas leave the Service's endpoints automatically – the gateway's own tracking makes canary infrastructure free.
3. **Failure semantics per route** – failOpen (engine down → gateway routes direct to pool via weight-0 fallback backendRef) or failClose (drop); explicit per WAFPolicy, never silently defaulted (R7 discipline).
4. **Protocols** – h2c engine ingress; gRPC framing recognised; TLS termination gateway-owned, so the engine inspects plaintext; visibility contracts per traffic shape are documented (kTLS/app-internal-TLS estates are a documented gap, with optional uprobes per workload – and honest admission in the docs).
5. **Binding** – policy attaches via HTTPRoute annotation trishula.security/policy: <name> or namespace-scoped WAFPolicy.routeSelectors; most-specific wins; Bound / UnresolvedRef status Conditions mirror Gateway API conventions.

A v2 shape removes the proxy hop entirely on Envoy-family estates (NGF, Envoy Gateway, kgateway, Istio): the engine's `ext_proc` server (shipped from Phase 0 in the same binary – one binary, two transports) inserts as a `ProcessingRequest` processor alongside the Inference Extension's `EndpointPicker`, so a WAF verdict and pool selection compose in one callout chain [43]. On BNK-class data planes today the documented path remains plain-backend steering; `ext_proc` activates where the data plane supports the callout chain.

3.3 The three deployables

- **trishula-shield** (DaemonSet, privileged, hostNetwork): loads and pins XDP/TC programs (cilium/eBPF Go loader, CO-RE objects compiled in-CI via `bpf2go` – a tagged release ships byte-identical eBPF objects), owns kernel enforcement, exports the ring buffer (16 MiB default) and BPFFS `/sys/fs/bpf/trishula/` pin dir. Preflight: kernel $\geq$ 5.8 with BTF on shield nodes; shield-less or degraded nodes fail loudly (Condition), never silently.
- **trishula-engine** (Deployment + Service): the userspace WAF; HPA on p99 inspect-latency plus ring depth; node-affined to shield nodes (BPFFS `hostPath` mount for the shared ring), topology-spread; each replica drains its node's ring directly – verdicts are content-keyed, so no per-flow affinity exists to preserve.
- **trishula-operator** (Deployment): watches CRDs, compiles bundles (evaluator chains + matcher DB + kernel map values + ban thresholds), distributes signed digest-pinned artifacts, manages shadow lifecycle and status Conditions, records FP-feedback telemetry.

3.4 Operating modes and the adoption ladder

Four modes cover the adoption ladder: **shield-only** (drop-in hardening of an existing path – zero proxy hop; the lowest-friction trial), **inline** (the v1 default for protected routes), **shadow-parallel** (engine evaluates and forwards regardless – CRS-import confidence, rule canary, ML learning, migration diffs versus an incumbent WAF), and **schema-learn** (API discovery emitting OpenAPI diffs for review). Every mode is per-route CRD state; the migration pattern from an incumbent WAF (say, the `nginx+ModSecurity` tier an estate already runs) is `shadow-parallel` → differential telemetry → per-route weight ramps at the gateway – no cutover weekend.

4. The Detection Stack

4.1 The ladder

The engine evaluates a seven-family ladder (S0–S7), short-circuiting on the first decisive verdict per the route’s action precedence (`block > challenge > log > pass`; SecLang `matcaction` semantics preserved for imported CRS rules):

- **S0, kernel verdict cache and ban table** – requests with a cached verdict or an active ban are resolved before the engine does any work (hit $\approx$ no-op; enforced at XDP/TC in the kernel without a userspace round-trip).
- **S1, Vectorscan matcher** – the hot subset of CRS + CEL patterns compiles to a vectorised scanning database (open-source Hyperscan family; x86/ARM/POWER; Suricata-backed [128], [129], [172]); a single scan across the whole set costs microseconds; block/streaming modes over reassembled bodies [136].
- **S2, CRS via the embedded reference evaluator** – Coraza-embedded OWASP CRS evaluation (SecLang import, paranoia-level profiles, per-rule exclusions) [139], [141], [178]: the correctness baseline for coverage, parity-checked per Section 4.6.
- **S3, CEL rules over the structured request view** – the developer’s own rules (Section 5.2).
- **S4, positive security** – OpenAPI schema enforcement and API discovery (Section 4.4).
- **S5, bot detection** – three detector families with consensus merging (Section 4.3).
- **S6, rate limiting and ban scoring** – per-route algorithms feeding ScoredWindowBan (Section 4.5).
- **S7, ML scoring** – in-band ONNX tier with async semantic-judge escalation (Section 4.7).

Every family emits verdicts in the same shape (matched-rule IDs, score contributions, evidence references), so the ladder composes into one verdict record (Section 5.3).

4.2 CRS as the correctness baseline – and the parity gate as the honesty mechanism

“CRS-compatible” claims die quietly in every engine in this field. Trishula makes them CI failures instead of marketing language. The **differential parity gate** runs the upstream CRS regression corpus plus curated adversarial corpora plus recorded production samples (go-ftw format [177], [165]) through both the Trishula path and the Coraza reference evaluator; the parity assertion is verdict-level (block/log/pass + matched rule IDs + score ladder), it blocks release at 100% applicable rules (M1 in the PRD’s metric table), and in production it runs shadow-parallel against the incumbent WAF during a migration, with any FP delta or missed match generating an exclusion/tuning workflow item rather than a shrug. A fast-path pattern the reference evaluator disagrees on is demoted automatically – self-healing parity, not a snapshot test.

4.3 Bot detection as evidence accumulation

Bots are the workload most HTTP security products still treat as a signature problem. Trishula treats them as **evidence accumulation** across three independent detector families: **TLS fingerprints** (JA4/JA4S at the termination hop [197], [F-ja4]), **header-plane fingerprints** (header order, casing, value entropy, HTTP-version quirks – ordered-hash comparison against browser/tooling profiles), and **behavioural windows** (kernel-observed pacing, path-traversal graphs, session continuity, auth-failure ratios – the harder-to-forge altitude precisely because it is kernel-observed rather than client-controlled). Each detector emits (`detector_id`, `confidence`, `evidence_ref`) – never a bare boolean – and the merge policy is per-`WAFPolicy` configurable consensus (≥ 2 families agree to act; single-detector matches log and score). Known-good bots (documented crawlers, health checkers) are an allowlist CR with reverse-DNS validation on termination-hop identity – misclassified-good bots are a tuning case, not a detection failure.

Honest limits stated in the docs: TLS fingerprints are forgeable by a motivated adversary (JA4 is a cost-raiser, not an identity); fingerprint databases decay as tooling evolves (the known-tooling matching set ships as a digest-pinned feed CR on a refresh cadence, not hardcoded); and the design assumes adversaries adapt, which is why the behavioural altitude carries the arbitration weight.

4.4 Positive security: OpenAPI as policy, discovery as onboarding

The tier-1 commercial feature the OSS field rarely ships. An `APISpec` CR wraps the OpenAPI 3.1 document the API team already maintains (per host and route-prefix); **enforce mode** blocks unmatched paths/methods and per-field violations (type, length, pattern, required) with diagnostics naming the violated field; **learn mode** builds the spec from live traffic (endpoint extraction, JSON-schema inference with cardinality tracking), emits CR diffs for review, and surfaces drift telemetry from the moment enforce switches on. The lifecycle mirrors everything else in Trishula – learn → shadow → enforce, evidence-gated – and derives security posture from an artifact developers maintain for other reasons anyway, which is the adoption thesis applied to detection: the spec they wrote for their SDK is the policy.

4.5 The ban algorithm: ScoredWindowBan v1, specified

The named differentiator – and an example of the specification discipline the product applies to algorithms. `ScoredWindowBan` is `fail2ban`'s operational UX [147] re-specified for HTTP: computed on the wire, enforced in kernel maps, observable as first-class OTel events, and version-pinned by name with a conformance test any implementation must pass to claim it (a spec-reproduction test ships from day one; changes to the algorithm require a new version, never a silent rewrite).

State per source key `s`: an evidence window `events[s]` of (timestamp, weight) pairs; a ban state `ACTIVE_BAN(until, tier, reason_codes)` or `CLEAR`; and a recidivism counter. Constants are `BanPolicy` CR knobs with `fail2ban`-vocabulary defaults: `findtime W = 10m` (evidence window), `threshold T = 10` (score, not raw count), `bantime B = 10m`, `backoff K = 2` (exponential repeat multiplier), `maxBantime = 24h` (permanence is a decision, not an accident). Scoring weights are CR-configurable: CRS critical match +5, CRS warning +2, bot-consensus verdict +4, single-detector +1, rate-window breach +3, schema violation +2, auth failure +2, kernel EWMA burst flag +3; scores decay exponentially (half-life 20% of `W` – staleness by design, so a single old event cannot carry a ban). The decision loop:

$score[s] = \sum w_i \cdot \exp(-\lambda(t_{now} - t_i))$ over the window; on $score \geq T$ the engine issues `ACTIVE_BAN` with `until = t_{now} + \min(B \cdot K^{\text{recidivism}}, B_{max})`, emits the ban event carrying the *score trajectory* (the contributing events with weights) as the evidence record, and writes the key into the kernel `ban_table` – after which XDP/TC enforces in $\sim 1 \mu s$ without consulting userspace. Unban is automatic at expiry; analyst release is `trishulactl ban release` (RBAC-audited); and the developer-release path – a `BanRelease` CR an application team applies for their own route class during a false-positive incident – keeps the victim of a ban on their own Kubernetes-native unban path, no console required.

Two safety gates are built in: **prefix escalation** (banning a /24 when $\geq N$ distinct keys from it ban out within `W` – the CGNAT/office-NAT collateral-damage failure mode of every ban system) is implemented but **off by default**, gated behind a threshold designed to make accidents rare; and per-identity rate keys (`JWT_sub / client_id` when identity is present) score the actor rather than the address whenever the protocol provides identity. Ban false-positive rate in shadow mode is a published metric with a budget and an alert (`BanRateAboveBudget`), and post-release recidivism tracking distinguishes “oops, CI ran from a datacenter IP” from “retrying botnet node.”

4.6 Rate limiting (the limits half of limits-and-bans)

Four algorithms are `BanPolicy` /route-class selectable: **token bucket** (steady-state per-route budgets; the default for API quotas, NGINX/Envoy-semantics-compatible [149], [F-`rfc9110`]), **sliding window** (exact windowed count over `k` samples; anti-brute-force on auth endpoints), **EWMA rate** (exponentially weighted arrival per source – kernel-side counters already produce these), and the **score ladder** itself (the ban decision). Rate keys compose per route: source IP (default), IP + route class, JWT identity when present, JA4 cluster. Every 429 Trishula emits carries `Retry-After` – a deliberate developer-experience signature: a rate-limited client learns exactly when to retry, from the component that rate-limited it.

4.7 The ML tier with a clean hand-off contract

ML never reads packets. The binding design rule: ML consumes versioned, structured feature views derived from engine-parsed requests plus eBPF-sourced counters – the contract that makes ML auditable (a verdict can name its

feature vector and model version), testable (a FeatureView is an artifact with a schema), and evolvable (a model change is a digest-pinned CR, shadow-first like everything else).

Four tiers, cheap first: **T1**, an in-band ONNX scorer on CPU for tree/linear models over tabular features (μ s-class inline; the hard budget is < 1 ms p99, with auto-degrade to async under backpressure – the request is never queued on ML); **T2**, a verdict cache (userspace LRU keyed on normalised request signature, TTL + model-version tag) whose high-confidence verdicts and ban scores write back to kernel maps – closing the loop with the kernel plane; **T3**, an escalation pod (Triton [242]) for suspicious-band traffic only, synchronous only when the route budget allows, else async with cache backfill; **T4**, an async semantic judge (LLM-class) for prompt-injection/jailbreak/exfiltration semantics – always asynchronous, suspicious-only, verdicts cached, judge model and prompt pinned as supply-chain artifacts, heterogeneous judges for robustness (the judge is itself an attack surface [217], [218], [219]), oracle access rate-limited, and verdicts **never LLM-only** (the deterministic families' corroboration is policy-checked).

The evasion posture is documented rather than implied: mutation-based evasion and surrogate-model optimisation are the assumed adversary [230], [234], [211]; mitigations are probe rate-limits (bounding oracle budget on the expensive tiers), shadow-first learning inputs, perturbability-scored feature selection (favouring hard-to-perturb flow/behavioural statistics over user-controlled strings), and heterogeneous ensembles. The honest claim is **raising attacker cost to impractical, not purity** – no ML WAF is evasion-proof, and the product does not pretend otherwise; drift monitoring on per-endpoint verdict distributions doubles as an **evasion canary** (a rise in suspicious-but-allowed traffic is often probing), with retraining gated off confirmed drift rather than raw alerts [243].

5. Developer Experience: the Charter, the OTel Narration, the Curriculum

5.1 A charter enforced by CI

Adoptability is a testable property, not a hope. The charter's commitments each carry a CI check or acceptance test – and a charter violation is a bug with the same severity as a detection miss, *because the thesis is that understandability is the security feature*:

Table 2: The developer-experience charter (each commitment CI-enforced)

#	Commitment	Test that enforces it
DX1	First custom rule ships ≤ 15 min from <code>git clone</code> (kind + NGF + Trishula + one CEL rule lab)	CI runs the lab nightly; timing asserted (M3)
DX2	Every CEL error names the offending field and suggests the structured-view path	fixture corpus $\rightarrow$ assert error text shape
DX3	Every documented rule links to the wire-level explainer for the fields it touches	docs lint: backlink check per example
DX4	No rule requires vendor-dialect knowledge (no ModSec directives exposed unless importing CRS)	docs lint + API review gate
DX5	Every block verdict self-explains: rule ID, plain-language reason, link to the rule's doc + fixture request	golden-trace tests over verdict telemetry
DX6	Uninstall returns the cluster to pre-install state, zero kernel residue	uninstall CI job runs <code>bpftool assertions</code> (M7)
DX7	All docs examples run – fixtures are CI-executed, never decorative	fixture runner over docs corpus (M4)
DX8	One config surface: CRDs + OpenAPI – the spec written for SDK docs is the policy	schema-validate APISpec CR against its OpenAPI
DX9	<code>kubectl apply</code> is the entire deployment story – no console, no bespoke REST	API review gate: new features must ship as CRD surface
DX10	Every 429 carries <code>Retry-After</code>	response lint on the ban/rate paths

The developer's day, in the charter's vocabulary: **Day 0**, one Helm install in the kind lab; the walkthrough is the documented proof. **Day 1**: point Trishula at an API the developer already has an OpenAPI doc for; observe shadow verdicts in the OTel backend the team already runs; promote to enforce on evidence. **Day 2**: own it – rule tweaks, excludes, ban thresholds as reviewable PRs over the same CRDs; no console appears anywhere, and the OTel backend is the console.

5.2 CEL as the rule language (and why not a new DSL)

Trishula deliberately ships no rule DSL. The rule language is **CEL** (Common Expression Language [137]) – the small, well-specified expression language already used in Kubernetes admission control and Envoy configuration [157], [146] – evaluated over a **structured request view**: `request.method/path/headers/body.{json,form,text}` (typed accessors), `request.client.{ip,asn,geo,reputation}`, `request.ja4`, rate windows, response view, and the transaction state (`tx.score`, matched rules, semantic flags such as `smuggling_suspect`). A rate-shape rule is two lines; a body-depth guard for a chat endpoint is three. Choosing CEL over Rego/proxy-wasm/new-DSL: it is small and formally specified; it is Envoy- and K8s-familiar (developer knowledge transfers); it compiles to fast AST evaluation with a *deterministic cost model* – per-rule cost budgets make rule-DoS structurally impossible from day one.

5.3 Every verdict narrated: OTel as first-class product surface

Observability is a design commitment rather than a retrofit, and it is the delivery mechanism for the developer-first thesis: **OTLP traces, metrics, and logs as equal first-class signals** [F-otel]. One server span per request across all

stages (parse → ladder → verdict → forward); shield events attach as span links; per-rule match counters, per-phase latency histograms, ring depth, cache hit rates, ban gauges and score trajectories, ML tier latencies, schema-drift counters, exclusion hit counts – exemplars carry `trace_id` so a dashboard anomaly jumps straight to the trace; and every verdict emits a structured **OTLP log record** in a versioned schema (`trishula.verdict.v1`): the full ladder walk (per-stage matches and microseconds), the plain-language reason, the doc-reference link, and for bans the complete evidence block (the score trajectory). Redaction is structural, not optional: authorisation/cookie/token-shaped header values are dropped at emission; body samples never leave the pod without a per-policy flag plus purpose binding. The developer debugging a block reads it in the telemetry stack they already run.

5.4 Teach the web (the curriculum as a product deliverable)

The SPIFFE/SPIRE lesson: the project that *explains why the wires work* becomes the reference implementation people trust. Trishula’s documentation ships a five-module curriculum – **the wire** (HTTP/1.1 framing, HTTP/2 streams/HPACK, smuggling: why the parser is strict), **TLS and QUIC** (handshakes, ALPN, JA4 survival-of-encryption, 0-RTT risk), **eBPF as a platform** (XDP/TC hooks, maps, ring buffers, verifier bounds, and why L7 does not live in kernel), **security modelling** (negative vs. positive security, anomaly scoring and paranoia levels, fingerprints vs. behaviour, rate algorithms and ban ladders), and **the frontier** (HTTP/3 posture, post-quantum key exchange and what a WAF must not assume under future TLS revisions, AI request/response abuse classes) – maintained in-repo with the same CI as product code, with every documented rule example linking to the bytes it inspects (DX3), and the verdict’s own telemetry carrying the link at incident time (DX5) – which is when developers actually read. Adversarial corpora (the smuggling corpus, the CRS differential suite) double as teaching fixtures: the lab and the lesson are the same cluster.

6. Community, Governance, and the Rules Question

6.1 The footprint

The project's public footprint is the GitHub organisation `trishula-dev` (github.com/trishula-dev) with the domain `trishula.dev` for docs/site; the upstream `trishula` GitHub handle is a dormant 2013 account (verified at the time of writing) – the `-dev` suffix is permanent and no aliasing layer is planned around it. The repo split mirrors the governance boundary: `trishula` (the product repo: Apache-2.0; engine, shield, operator, charts, tests, labs) and `website` (the docs site). Curated rules ship **in-repo** (`rules/cel/`, `rules/exclusions/`) – every shipped rule parity-tested and versioned with the engine that runs it.

6.2 Community-sourced rules, gated honestly

The user-requested corpus question – should a `rules` repo exist for crowd-sourced CEL rules or faster rule authoring? – resolves to a **graduation discipline** rather than a day-one split. The reasoning: the product's correctness guarantee (the differential parity gate, Section 4.6) is only as strong as the rules being tested against the same engine revision they will run on. A community rule corpus that ships *with* the product repo inherits the parity gate for free; a corpus that ships in an out-of-tree fork does not. Therefore: **Day 0-Phase 1** – curated shipped rules live in-repo, community contributions land as ordinary PRs into `rules/`, and every contribution travels the same shadow-first → parity-tested → evidence-gated ladder any other change takes. **Phase 1+** – if external contributions justify the review surface (volume, contributor diversity, moderation load), the curated set graduates into a separate `trishula-dev/rules` repo, with the *quality gate moving with it*: a CI lane that runs the graduated corpus against a pinned engine version set keeps the parity guarantee intact while allowing the community surface to iterate on its own cadence. The design decision is recorded in the PRD (§2.7 D6) as a standing position rather than an open question.

6.3 Governance and supply-chain posture

The product repo stays Apache-2.0 code-only (clean provenance for packagers and adopters); contributions require the CI gates plus maintainer review, commits are signed, and no commit lands directly on a main branch. The supply-chain posture carries from the threat model: rule bundles digest-pinned and operator-signed; MLModel refs digest-pinned; CRS bundles from pinned upstream tags with recorded hashes; kernel maps restricted to the shield's UID with foreign-map-writer alerts. Trust boundaries are documented: what Trishula trusts (the gateway's TLS termination and steering marks, the BPF verifier, the node kernel, OCI registry integrity, CRD RBAC below the operator) and the residual risks – documented, not hidden.

7. Performance: Targets, Stated as Targets

Honesty about performance is a design decision: performance claims are labelled *targets* until benchmarked, and the benchmark plan is part of the product. The headline budgets (per-request p99, full CRS + 8 KB body): ≤ 1.5 ms added inline end-to-end (≤ 300 μ s p50); XDP shield hit-path ≤ 10 μ s; kernel-cached verdict/ban short-circuit ≈ 1 μ s kernel-side; throughput ≥ 10 k req/s per 4 vCPU engine replica; shield drop ≥ 10 Mpps/core on a commodity 25 G NIC. The kernel-side numbers rest on literature-shielded envelopes – Cloudflare’s measured 10 Mpps/core XDP drop [60], the FAST’18 paper’s 24 Mpps/core [75], AF_XDP zero-copy measurements – not vendor claims. The target ladder escalates deliberately: **Phase 0 (PoC)** ships the honest exit criteria of ≤ 10 ms inline unoptimized with 100% verdict telemetry and the CRD round-trip proven; **Alpha** hardens to the full budget. The harness (`bench/` in-repo: `wrk2` closed-loop, `h2load`, `vegeta` burst shapes, flood scripts) runs in CI with p99-regression gates; published results carry harness configs so they are rerunnable, and any claimed number that has not been measured reads “target” in every artifact it appears in.

8. Threat Model of Trishula Itself

Security infrastructure whose own compromise is an escalation path deserves its own model (PRD §18). The treatments worth naming in a paper-length summary: **parser-differential bypass** (smuggling-shaped requests that Trishula’s parser normalises differently than the upstream pool) – treated by strict RFC 9112 normalisation, reject-on-ambiguity, and a smuggling corpus in the differential suite; **shield blind spots as bypass windows** – atomic per-hook program replacement and readied-node gating; **BPFFS mount abuse** (a privileged pod poisoning ban tables) – pins confined to the trishula-owned subtree, map-write permissions restricted to the shield UID, alerts on foreign map writers; **ban-table weaponisation** (an adversary farming evidence against a victim’s key on shared egress) – prefix escalation off by default, per-identity keys where the protocol provides identity, and the `BanRelease` path as the documented victim recourse; **DoS against Trishula** (regex-catastrophic bodies, schema bombs, h2 rapid-reset) – parser admission caps, Vectorscan’s no-backtracking guarantee, per-request resource budgets, settings caps; **debug samples as data leaks** – the redaction pipeline (Section 5.3); and **fail-open regression** – schema-required explicit `failureMode`, preflight admission checks, and drift alerts when an enforce-mode route lacks healthy engine coverage. The vantage problem is stated plainly: the engine sees tenant payloads, so egress is network-policy confined, telemetry exports pass the same redaction, and the ban ledger is access-controlled like an audit log.

9. Limitations and Future Work

PoC-first honesty. No implementation exists at the time of writing: the PRD is complete and reviewed; the PoC starts this sprint. Every performance figure in Section 7 is a target pending the benchmark plan’s measurements, and the project’s README commits to an “exists vs. doesn’t exist” table from commit zero – the honesty mechanisms are product features, applied to the project itself.

The parity gate’s corpus is the load-bearing artefact. Verdict-level parity is only as good as the corpus it runs on; upstream CRS regression tests are necessary but insufficient, and the curated adversarial + recorded-production corpora (the smugglers’ corpus, FP-heavy realistic mixes) are a maintained asset rather than a checkbox. The gate’s honesty depends on it staying adversarial.

Vectorscan scale and memory. The compiled matcher’s memory at CRS-scale (roughly hundreds of MB for the full hot subset) is budgeted (R2b) and CI-validated, with subset-selection by frequency as the pressure valve; the differential gate bounds the behavioural risk.

kTLS and app-internal TLS remain blind. Where termination happens inside an application pod (no proxy hop, no uprobes), the engine sees nothing – a documented visibility contract per traffic shape, with uprobes as the opt-in per-workload restoration. Shield altitude (L3/L4) still applies.

CGNAT and shared-egress collateral. Source-IP-keyed scoring on shared egress risks collective punishment; the mitigations (identity keys, prefix-escalation default-off, BanRelease, evidence records) reduce but do not eliminate the failure mode, and the steady-state FP budget is the published accountability line.

Where the roadmap does not go (v1): no TLS-terminating edge load-balancer (the gateway owns termination), no service mesh (east-west stays with the CNI), no secrets/token vault, no endpoint picker (model-pool selection stays with EPP/the gateway), no vendor console, no Windows or non-Linux kernel ports, and no HTTP/3 data-plane claims (tracked evaluation with a curriculum module teaching why, before any claim).

Phasing. Phase 0 (this sprint, 8 weeks): the shield PoC, ring-drain engine, HTTP/1.1 ladder, NGF steering lab, CRS parity v0, the operator compile round-trip, ScoredWindowBan in shadow, OTel pipeline, the honest README + demo script. Alpha (months 2–4): performance hardening, Vectorscan with differential gate, positive-security learn mode, bot detectors, ban shadow→enforce with `BanRelease`, the first academy modules, and a secondary-gateway CI lane (Envoy Gateway) to keep the portability contract honest. GA track and beyond: ML enforcement with promotion workflows, behavioural-DoS detectors, `ext_proc` activation on Envoy-family estates, DPU evaluation, OpenShift/multi-arch, HTTP/3 posture evaluation, response-side exfiltration guard.

10. Conclusion

Trishula is an argument about where the WAF's next decade of adoption can come from – stated as a design, not yet as a result. The argument's claims: that the crossing the WAF never made happened for every other security control when configuration became declarative, reviewable, and developer-owned, and that CRDs-compiled-to-kernel-and-userspace-altitudes is the WAF's version of that crossing; that the OSS field's missing depth (positive security, bot defence, specified temporary bans, kernel altitude with parity discipline) is buildable as one compiled artifact rather than a feature list spread across disjoint projects; that a named, versioned, conformance-testable ban algorithm does for temporary bans what CRS did for signatures – makes the mechanism discussable, reproducible, and improvable in the open; that honesty mechanisms (shadow-first promotion, differential parity, targets-labelled-as-targets) are load-bearing engineering choices rather than documentation hygiene; and that the developer experience – a first rule in fifteen minutes, a self-explaining verdict carrying a link to the lesson it teaches, security answers in the telemetry stack already open – is the adoption path, because understandability is the security feature.

The work ahead is the PoC this sprint and the honest checklist that follows it. The design's own threat model applies to the project: the differential gate is the mechanism that keeps "CRS-compatible" from becoming a market claim, and the target label on every unbenchmarked number is the mechanism that keeps the performance story honest. Trishula is a work in progress at the trishula-dev organisation; the full product specification lives in the companion PRD [PRD], the build board's issues graduate to the public tracker when the repos land, and the curriculum begins on trishula.dev as the docs ship with the code they teach.

11. References

- [1] Cloudflare. “How to drop 10 million packets.” blog.cloudflare.com/how-to-drop-10-million-packets. (XDP drop rates ≥ 10 Mpps/core measured in production.)
- [2] Cloudflare. “L4Drop: XDP DDoS mitigations.” blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations. (XDP L3/L4 mitigation lineage.)
- [3] Cloudflare. “xdpcap: A tcpdump-compatible XDP capture tool.” blog.cloudflare.com/xdpcap. (Kernel-dropped-packet forensics.)
- [4] Xiao, Zhihui et al. FAST '18. “The Art of Network Shaping: Tailoring XDP to the Packet Processing.” dl.acm.org/doi/10.1145/3281411.3281443. (24 Mpps/core, 41.6 ns/packet XDP.)
- [5] Meta. “Katran: a scalable network load balancer.” github.com/facebookincubator/katran. (Production XDP load balancing.)
- [6] Pixie Labs. “eBPF OpenSSL Tracing.” blog.px.dev/ebpf-openssl-tracing. (Uprobe-based TLS visibility; the ciphertext-at-NIC problem.)
- [7] Cilium. “Cilium architecture” and the Envoy L7 proxy integration. docs.cilium.io. (The two-plane pattern: eBPF fast path + userspace L7 proxy.)
- [8] cilium/ebpf. github.com/cilium/ebpf. (Go eBPF loader library; ringbuf, maps, bpf2go.)
- [9] Nakryiko, Andrii. “BPF Portability and CO-RE.” nakryiko.com/posts/bpf-portability-and-co-re.
- [10] eBPF docs. “Verifier” and “Tail calls.” docs.ebpf.io. (Instruction budgets, bounded loops.)
- [11] ebpf-go guides. ebpf-go.dev/guides/getting-started.
- [12] VectorCamp. “Vectorscan.” github.com/VectorCamp/vectorscan. (Open-source Hyperscan family; x86/ARM/POWER.)
- [13] Suricata/OISF. “Why Vectorscan matters to Suricata.” suricata.io, 2026.
- [14] Intel. “Hyperscan” (last open-source release 5.4.2). github.com/intel/hyperscan.
- [15] OWASP Core Rule Set. github.com/coreruleset/coreruleset. (CRS v4.x; SecLang; paranoia levels.)
- [16] OWASP CRS. “Announcing CRS v4.25.0 LTS.” coreruleset.org, 2026.
- [17] OWASP CRS. “Migrating CRS 3 to 4.” coreruleset.org, 2026. (Overview and FP-tuning parts.)
- [18] Coraza. coraza.io · github.com/corazawaf/coraza. (Go CRS-compatible WAF engine; 100% CRS v4 compatibility posture; WASM plugins.)
- [19] go-ftw. “CRS regression testing framework.” github.com/coreruleset/go-ftw.
- [20] CRS plugin registry. github.com/coreruleset/plugin-registry.
- [21] ModSecurity stewardship: Trustwave→OWASP transfer (owasp.org/blog, 2024); nginx/NGINX-AppProtect EOL trajectory (nginx.com/blog; F5). (The engine-stagnation cautionary tale.)
- [22] System Shield Hardening. “WAF rule tuning” (4–8 h per-endpoint exclusion cost). systemshardening.com.
- [23] CEL. “Common Expression Language specification.” github.com/google/cel-spec.
- [24] Kubernetes. “CEL – the Common Expression Language.” kubernetes.io/docs/reference/using-api/cel.
- [25] Envoy. “Matching API” and CEL matcher documentation. envoyproxy.io/docs/envoy.
- [26] Envoy. “Local rate limit filter proto.” envoyproxy.io/docs/envoy/main/api/envoy/extensions/filters/http/local_ratelimit/v3.
- [27] GoogleGroup fail2ban documentation. fail2ban.readthedocs.io. (findtime/maxretry/bantime/recidive vocabulary; bansubnet.)
- [28] NGINX. “ngx_http_limit_req_module.” nginx.org/en/docs/http. (Rate-limit semantics.)

- [29] RFC 9110: HTTP Semantics. www.rfc-editor.org/rfc/rfc9110. (429 + Retry-After.)
- [30] NGINX Gateway Fabric 2.7 release (Gateway API 1.6 conformance, TLS Terminate, external auth). [blog.nginx.org, 2026](https://blog.nginx.org/2026).
- [31] NGINX Gateway Fabric WAF integration documentation. docs.nginx.com/nginx-gateway-fabric/waf-integration. (WAFPolicy attachment and compiled-bundle patterns; securityLogs.)
- [32] F5 WAF for NGINX overview. docs.nginx.com/waf/fundamentals/overview.
- [33] Kubernetes SIG Network. “Gateway API Inference Extension.” gateway-api-inference-extension.sigs.k8s.io. (EPP; the v1 API reference.)
- [34] GIE. “EndpointPicker Protocol – proposal 004.” github.com/kubernetes-sigs/gateway-api-inference-extension, v1.4.0 tag. (ext_proc protocol and failure semantics: FailOpen/FailClose.)
- [35] open-appsec. github.com/openappsec/openappsec; docs.openappsec.io. (Learning→enforcement lifecycle; central management plane; API-schema enforcement.)
- [36] SafeLine (Chaitin). github.com/chaitin/SafeLine; help.waf-ce.chaitin.cn. (Staged semantic pipeline; docker-compose shape.)
- [37] FoxIO-LLC. “JA4/JA4S TLS fingerprint specification.” github.com/FoxIO-LLC/ja4.
- [38] Wallarm. “API abuse prevention.” docs.wallarm.com. (Behavioural fingerprinting, credential-stuffing posture.)
- [39] CrowdSec. docs.crowdsec.net. (AppSec/bot detection; community blocklists; bouncers.)
- [40] Synapse (FoxIO/gen0sec). github.com/FoxIO-LLC/synapse. (XDP NDR + inline proxy; JA4+ blocking; Wirefilter.)
- [41] Moat (arxignis). github.com/arxignis/moat. (Rust XDP edge proxy; JA4+ suite.)
- [42] MicroWAF (dcc-bigfred). github.com/dcc-bigfred/microwaf. (Host WAF; CEL rules engine; XDP drop rates.)
- [43] Spliff. news.ycombinator.com/item?id=46663319. (XDP + uprobes decrypted-TLS correlation.)
- [44] TEMPEST-BREAKER. “Defeating volumetric DDoS at the network edge” (Rust AF_XDP tarpit). medium.com/@okywowww.
- [45] BunkerWeb. github.com/bunkerity/bunkerweb; NAXSI (wargio) github.com/wargio/naxsi; Janusec janusec.github.io; Apache APISIX apisix.apache.org; Higress github.com/higress-group/higress. (The surrounding field’s shapes.)
- [46] OpenTelemetry. opentelemetry.io. (OTLP traces/metrics/logs; the narration surface.)
- [47] OpenReview. “Adversarial evasion of ML classifiers.” openreview.net/pdf?id=m5AntlhJ7Z5; and the evasion-survey arXiv corpus (2506.09443; 2306.05494; 2312.13041). (ML-WAF evasion literature.)
- [48] Triton Inference Server. github.com/triton-inference-server/server. (Model escalation tier.)
- [49] Young, Juho. “AI platform ML model monitoring/drift detection (Evidently, MLflow).” youngju.dev, 2026. (Drift tooling posture.)
- [50] Trishula – Product Requirements Document & Project Charter (v4). trishula-dev project internal document, October 2026. §1–§21, Appendices A–E. (The full specification; this paper’s companion.)