ONRouter: An OpenResty-Compatible Gateway in Rust with a Preemptible Lua Runtime

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com
https://github.com/yxpil/ONRouter

Abstract

ONRouter (ORR) is a reverse proxy and application gateway written from scratch in Rust that reimplements the OpenResty programming model — nginx's multi-phase request pipeline, an embedded Lua 5.4 scripting layer exposing the `ngx.*` API, native Rust modules, and TLS termination — without reusing any gateway logic from nginx. This paper describes the architecture and reports an on-host A/B comparison against OpenResty 1.25.3.1 across four scenarios and five concurrency levels. Three design decisions carry most of the result. First, a per-core Lua VM pool driven by a coroutine scheduler in which every resume is armed with an instruction-count hook that yields the coroutine back to the scheduler, so a runaway script in a single-process gateway degrades to one busy core rather than stalling the gateway. Second, an explicit blocking analysis that moved every filesystem access off the async workers and added an in-process static-file cache; this is where the static path win comes from. Third, a front-proxy seam placed strictly after the real-IP trust gate, so that a client replaying a provider's entire header set cannot move its own identity. The measurements are reported together with their caveats: the load generator is a Python client, so absolute request rates are client-bound; the Lua paths are *slower* than OpenResty's LuaJIT (0.74x at one connection, 0.99x at thirty-two), while the reverse proxy and static paths reach 2.49x and 1.82x. A control endpoint that performs no I/O shows parity (0.99x), which isolates the difference to the I/O and scheduling path rather than to the harness.

Artifact. The gateway, its configuration corpus, the benchmark harness, and the security test matrices are open source at `https://github.com/yxpil/ONRouter`. Every number in this paper is reproducible from the repository; the figure scripts are included with this submission.

1 Introduction

OpenResty has become the default application platform for nginx: it embeds LuaJIT inside nginx workers and exposes the `ngx` API Zhang (2014); Pall (2026); OpenResty (2026), so that rate limiting, routing, caching, and WAF logic can be written in a scripting language rather than in C or in nginx's directive vocabulary. The benefit is large, and the cost is equally well known: the platform is C, the scripting runtime is a single-threaded JIT inside a worker process, and the whole execution model is one process per worker with a fork-based reload. Safety guarantees are whatever C gives you Jung et al. (2018), and the deployment story assumes a POSIX process supervisor.

This paper describes **ONRouter**, a gateway that keeps the OpenResty programming model — the same phase pipeline, the same `ngx.*` surface, the same configuration dialect — but implements it in Rust on an async runtime. The goal is not to claim that Rust makes gateways faster. The goal is to ask a narrower question: *if you re-implement the OpenResty model from scratch in a memory-safe language, what do you have to redesign, and what does the redesign cost?* The answer turned out to be interesting in three places: the Lua concurrency model, the boundary between blocking and non-blocking I/O, and the trust ordering between edge-proxy metadata and client identity.

1.1 Contributions

C1. A per-core Lua VM pool with preemptive, coroutine-scheduled execution (§3.2). ONRouter runs one independent Lua 5.4 VM per logical core, mirroring OpenResty's one-VM-per-worker. Within a VM, a dedicated scheduler task interleaves many in-flight requests as Lua coroutines, and every resume is armed with a VM instruction-count hook, so a script that never yields is interrupted rather than allowed to monopolise its core. We describe the fast path that avoids a scheduler hop entirely for non-yielding scripts, and the accounting that makes the two paths observable.

C2. An explicit blocking analysis for the static and scripting paths (§3.3). Because a Rust async runtime makes blocking calls a correctness hazard rather than merely a performance one, we enumerate every filesystem access on the request path, move it off the async workers, and add a process-level cache for Lua sources and a size-bounded in-memory cache for static files with streaming fallback above the threshold.

C3. A trust-ordered front-proxy seam (§3.5). Metadata derived from CDN headers is only honoured when the direct TCP peer falls inside an operator-declared trust range, and the ordering is enforced by construction: recognition happens after the real-IP gate, and the mechanism that would let a profile promote itself to an authority requires an explicit opt-in that names the virtual host at configuration-test time.

C4. An on-host A/B evaluation with an explicit control (§5). We compare against OpenResty 1.25.3.1 on the same machine, with the same configuration, the same static file, the same Lua source, and the same Python backend. We include a control endpoint that does no I/O and show that it reaches parity (0.99x), which is what allows us to attribute the other deltas to the implementation rather than to the harness. We report the negative result honestly: the Lua paths are slower than LuaJIT.

C5. An auditable security posture backed by reproducible matrices (§3.7). Three test matrices — 30 protocol-level checks, 36 checks replaying a documented audit finding against real traffic, and 27 checks against the remote-execution surface — are part of the repository and are the basis for the claims we make and the gaps we acknowledge.

1.2 Non-goals

We do not claim a feature-for-feature replacement of OpenResty. The repository maintains an explicit, status-first gap analysis (`docs/OPENRESTY_GAP_ANALYSIS.md`) that enumerates what is missing, and we summarise it in §6. In particular, ONRouter has no master/worker split and no `USR2` binary upgrade, and it uses interpreted Lua 5.4 rather than LuaJIT.

2 Background and Related Work

2.1 The nginx phase model

nginx processes a request through a fixed sequence of phases — rewrite, access, content, header filter, body filter, log — each of which may terminate the request early nginx (2026); Fielding et al. (2022a). `lua-nginx-module` extends this by allowing Lua to run in any phase, and by adding asynchronous primitives (cosockets, timers, semaphores) that let a script yield without blocking the worker. The phase model is the reason nginx configuration composes: an operator can reason about *where* a directive applies, not merely *what* it does.

ONRouter reimplements the phase pipeline in Rust. The configuration parser lives in `onrouter-config`; the request pipeline, the proxy, the upstream manager, and the Lua runtime live in `onrouter-server`.

2.2 Lua inside nginx

The two dominant deployments differ in the interpreter. OpenResty uses LuaJIT, a just-in-time compiler for Lua 5.1 with a well-optimised interpreter tier, and targets a machine-code-heavy workload. ONRouter embeds Lua 5.4 Ierusalimschy et al. (2020) through `mlua` mlua-rs (2026), an interpreted runtime with a register-based VM and a generational, incrementally collected heap.

This difference is the single largest performance factor in our results, and we do not try to hide it. The gap analysis in the repository predicts roughly 0.7–0.9x on Lua-only endpoints for the interpreted runtime, and our measurements in §5 land in that band, converging to parity as concurrency rises and the client becomes the bottleneck. Section 5 states the causal story: LuaJIT's advantage is largest when the script is the bottleneck (one connection, trivial work) and vanishes when the client is.

2.3 Async runtimes and blocking calls

The Rust async ecosystem has a well-known sharp edge: calling a blocking function inside a task stalls the whole worker thread, not merely the request Lerche (2026); Klabnik and Nichols (2019). The standard mitigation is a separate blocking pool (`spawn_blocking`), at the cost of a thread handoff. The alternative — avoiding the blocking call — is sometimes better. We resolved every filesystem access on the request path by asking which of the two applies, and the answer differed per path: §3.3 explains why a static file cache beat `spawn_blocking` for small files, and why a streaming body beat both for large ones.

2.4 Admission control and cache policy

Two of the subsystems in ONRouter draw on established results that we restate because they are not obvious: TinyLFU-style admission Einziger et al. (2017) for the in-memory static cache, and the power-of-two-choices heuristic Mitzenmacher (2001) for time-decayed upstream selection. We cite these in §3.4 and §3.3 respectively.

2.5 Edge metadata and client identity

CDN and edge-proxy providers conventionally let the edge set headers such as `X-Forwarded-For`, a country code, or a visitor identifier, and the origin is expected to honour them. The security literature's standard warning is that such headers are trivially forgeable by a direct client MITRE (2026b). Common practice is to trust them only behind a known proxy, but the ordering is often implemented as a configuration convention rather than as a structural property. ONRouter makes the ordering structural (§3.5).

2.6 HTTP request smuggling

Request smuggling arises from disagreement between a front and a back endpoint about message boundaries, usually via `Content-Length` versus `Transfer-Encoding` ambiguity MITRE (2026a); Fielding et al. (2022b). ONRouter's

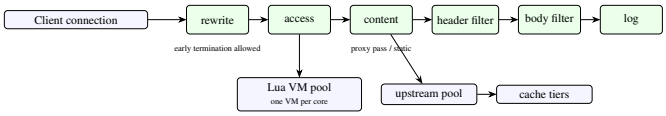


Figure 1: Request pipeline. The phase sequence mirrors nginx; Lua may run in any phase, and each phase may terminate the request early.

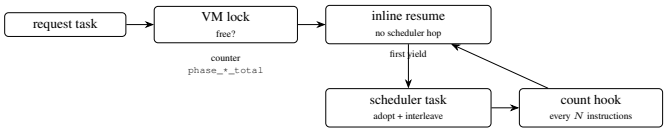


Figure 2: Lua execution paths. A non-yielding script completes inline; a yielding script is adopted by the VM's scheduler and preempted by the count hook.

defence is a normalisation-first approach: ambiguous framing is rejected with 400 rather than resolved by heuristic (§3.7).

3 Design

3.1 Overview

ONRouter is a two-crate Cargo workspace. `onrouter-config` parses the nginx-style configuration into a typed model. `onrouter-server` holds the request pipeline, the proxy, the upstream manager, the Lua runtime, and the subsystems (access control, rate limiting, caching, TLS, logging). The server crate is dominated by three files — `lua_engine.rs` (6,063 lines), `router.rs` (5,409), `server.rs` (4,650) — and the workspace totals 43,223 lines of Rust across 34 source files.

Figure 1 shows the phase pipeline. The implementation detail that matters for the rest of the paper is that the pipeline is a straight-line sequence of small units that can each yield control back to the runtime, so that an asynchronous Lua call in the content phase does not require the pipeline itself to be re-entrant.

3.2 The Lua runtime

This is the part of the system that differs most from the model it imitates, so we describe it in detail.

3.2.1 VM pool

ONRouter creates one independent Lua VM per logical CPU core, mirroring OpenResty's structure of one VM per worker. The VM selection is an atomic read of a round-robin cursor; there is no lock on the selection path. `init_by_lua` runs once per VM, matching nginx's once-per-worker initialisation semantics.

3.2.2 Coroutine scheduler

Within a VM, the model is not one request per VM. A dedicated scheduler task per VM interleaves many in-flight requests, each represented as a Lua coroutine. A request yields only at an explicit suspension point (`ngx.sleep`, a cosocket operation, a timer callback), and the VM lock is released between synchronous resume segments, so timers and other requests on the same VM continue to make progress. This is the single-worker-C10K behaviour of OpenResty, not one-request-per-VM.

3.2.3 Preemption

A single-process gateway has a property that multi-worker nginx does not: a script that never yields does not merely slow down its own worker, it parks that worker's thread outright. ONRouter therefore arms a Lua count hook on every resume. The hook fires after `lua_slice_instructions` VM instructions (default 1,000,000, approximately 10 ms), at which point the coroutine is yielded back to the scheduler; after eight consecutive slices the scheduler leaves its lock-and-await loop so that other work on the same VM can run. The mechanism is Lua's own debug count hook Ierusalimschy et al. (2020), not a modification of the interpreter.

3.2.4 Fast path

The scheduler hop is not free, and for the common case of a script that never yields it is not needed. When the target VM's lock is free, the request task compiles or resumes the first Lua segment inline; a script that never yields therefore completes without ever touching the scheduler task. Only on the first yield is the coroutine handed to the scheduler. A shared per-VM chunk cache keeps compilation single-flight. The split is observable at runtime through the counters `onrouter_lua_phase_{inline, adopted, scheduled}_total`; the repository's own traffic shows 100/100 inline for a no-yield script and a 50/50 inline/adopted split for a script that sleeps.

3.3 Blocking analysis and the static path

In an async runtime, a blocking call is a correctness hazard: it occupies a worker thread that is supposed to be multiplexing many connections. We enumerated the filesystem accesses on the request path and resolved each one according to its cost profile.

Lua source files. A script loaded from disk with `LuaSource::File` was read with a synchronous `fs::read_to_string` on every request — the single largest blocking point in the original implementation. It is now a process-level cache: a `OnceLock<RwLock<HashMap<PathBuf, Arc<str>>>>` that reads each file once, on first use. A shared per-VM chunk cache keeps the compiled form single-flight.

Static file resolution. Path resolution, index lookup, and autoindex generation now run inside

Table 1: Throughput on the static path, 32 concurrent keep-alive connections, before and after the blocking analysis. The *blocking analysis* column is the change from moving filesystem access off the async workers; the *static cache* column adds the in-process body cache on top.

Configuration	req/s	vs. baseline	source
Baseline	4,880	—	README
+ blocking analysis	6,715	+38%	README
+ in-process static cache	8,005	+64%	`bench_ab_32.log`

`tokio::task::spawn_blocking` on an owned snapshot of the request state, so worker threads are never occupied by disk I/O. Large body reads use `tokio::fs` chunked reads.

Error pages. Error-page target files are read once and the response built, removing synchronous round-trips from the error path.

Cache tiering. Files of at most 1 MB are held entirely in process memory, up to 512 entries, invalidated by a `(mtime, size)` pair. A cache hit costs one `stat` and no read; because cached bodies are `Bytes`, a whole-file hit and a `Range` slice served from the cache are reference-counted rather than copied. Files above the threshold, and `Range` windows that miss the cache, are *not* read into memory: `serve_static_flow` emits a chunked body that seeks once and reads 64 KiB per poll while the response is written, with an explicit `Content-Length` so keep-alive framing and `HEAD` remain exact — `HEAD` does not open the file at all.

Why not simply offload everything. Offloading the whole read to the blocking pool would have been simpler. It is wrong for two reasons. For small files, the hand-off costs more than the read: caching the body in memory removes both. For large files, buffering the whole body removes the memory bound: the repository's end-to-end test pauses a download mid-stream and observes a 0.6 MB resident-set delta for a 128 MB file, against whole-file buffering's 128 MB. The `sub_filter` directive is the one case that genuinely needs the whole body, so it buffers only when its MIME gate matches and only up to a hard 8 MiB ceiling; above that the file streams untouched and the skip is logged.

Admission policy. The in-memory cache admits entries under a TinyLFU-style frequency sketch with a bounded process-wide byte budget, so that a scan of unique URLs cannot evict the working set.

3.4 Upstream selection

The upstream manager offers three selection strategies with nginx-compatible semantics for the passive cases: a priority order, round robin, and a weighted order. On top of these, ONRouter implements an adaptive tier, `least_time`, which can be set to `off`, `observe`, or `on`.

The `on` mode uses the power-of-two-choices heuristic over a time-decayed latency estimate: two candidates are drawn, and the one with the lower decayed latency wins, with the draw charged to the winner. The decay means a node that was fast a minute ago no longer dominates. The `observe` mode computes the same statistic but never uses it to select, which makes the decision auditable after the fact; the repository's benchmark distinguishes `observe` from `on` by counting how often the two draws would have differed and how often a cold start occurred.

Passive health checking follows nginx token semantics for `proxy_next_upstream`: `error`, `timeout`, `invalid_header`, `http_XXX`, `non_idempotent`, and `off`, with separate `proxy_next_upstream_tries` and `_timeout` budgets. Connection-level reuse is tracked per upstream so that a failed connection is not handed to the next request.

3.5 Front proxy and client identity

The front-proxy layer recognises edge providers and derives metadata from their headers: the visitor address, the scheme, the country, and the visible host. The design constraint is that recognition must not be able to *grant* identity.

F1. Recognition happens *after* the real-IP trust gate. A value derived from a header is used only when the direct TCP peer falls inside a trust range the operator declared, or when `set_real_ip_from` applies. A client that copies a provider's entire header set is left on the node address for `allow`/`deny`, `limit_req`, `geo`, VIP reputation, the access log, and the cache key alike.

F2. Recognition alone only labels. A profile with no `probe` can never match.

F3. Promoting a profile's own ranges into a rewriting authority requires the explicit `front_proxy_auto_trust on`. It warns once per profile and names the virtual host at configuration-test time (`-t`).

F4. `Cache-Status` (RFC 9211) Nottingham (2022) is suppressed for unverified peers, so a hit rate that a client could inflate is not reported as a metric.

The repository contains a dedicated benchmark and proof script (`bench_front_proxy_20261020.json`) that drives real traffic through each of the four modes and records the resulting peer address and derived headers, so the ordering is verified by observation rather than by reading the code.

3.6 Still ONLine and the replay layer

The replay layer keeps a copy of a response so that the origin can be served during an outage. Because such a copy can be handed to a client that never fetched it, the rules are hard-coded rather than configurable: a response containing

`Set-Cookie` is never stored; a request carrying `Cookie` or `Authorization` never writes a copy, and by default is not served one either; `Set-Cookie` is stripped from every replayed response. Only anonymous `GET` requests with a 2xx response enter the zone, and the background re-fetch after recovery is itself an anonymous request that passes through the normal ACL, rate-limit, and rewrite path. This is a deliberate trade of freshness for availability: the replayed response may be stale, in the sense catalogued for non-transactional consistency models Viotti and Vukolić (2016), and the design accepts that in exchange for serving something at all during an outage.

3.7 Security posture

The security-relevant design decisions are listed in §5.5 together with the test matrix that backs each one. The principle is uniform: ambiguity is rejected rather than resolved, and access-control failure defaults to denial.

4 Implementation

ONRouter is written in Rust 2021 and builds on `tokio` for the async runtime, `hyper` hyp (2026) for HTTP/1.1 and HTTP/2, `mlua` for the Lua 5.4 runtime, and `rustls` for TLS. The workspace is two crates and 43,223 lines across 34 files; the configuration corpus under `conf/` contains 81 files, and `tools/` contains 82 end-to-end test scripts contributing 1,582 individual assertions. The Rust unit test suite contains 440 test functions.

The configuration dialect is nginx-compatible for the directive subset ONRouter implements. The compatibility surface is pinned by dedicated end-to-end scripts rather than by prose: `api_compat_e2e.py`, `include_e2e.py`, `auth_e2e.py`, `balancer_e2e.py`, and `compat_nginx.conf` together check that ONRouter and nginx agree on status codes, headers, and body bytes for the same configuration.

5 Evaluation

5.1 Methodology

The comparison is an on-host A/B. Both servers run on the same machine, with the same configuration semantics, the same static file, the same Lua source, and the same Python backend. The load generator is a Python program using `http.client` over keep-alive connections, with the number of client processes capped at half the logical core count so that the client does not starve the server. The sweep runs each configuration at 1, 4, 8, 16, and 32 concurrent connections, and each cell is run with a request count of at least `max(2000, connections × 400)`; all cells reported zero errors.

Three properties of the harness matter for interpreting the results.

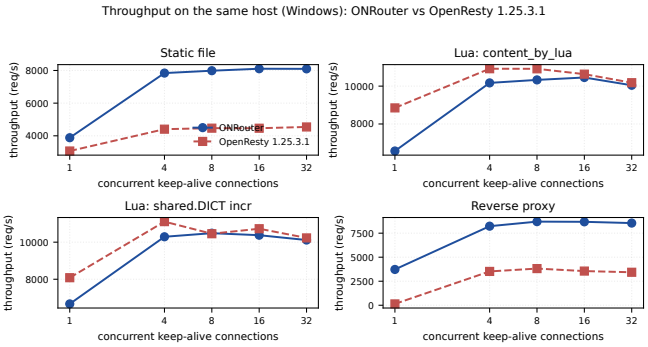


Figure 3: Throughput at five concurrency levels. Both servers share the host, the configuration, the static file, the Lua source, and the backend. The load generator's own ceiling is 9,591 req/s.

The client is the ceiling. The harness measures its own maximum sustainable request rate and prints it as a reference line (9,591 req/s in the sweep reported here). At the higher concurrency levels the measured server rates approach this ceiling, so the deltas between servers compress. This is the reason the per-scenario ratios are not constant across concurrency, and it is the reason we treat the high-concurrency numbers as client-bound rather than as server capacity.

There is a control. One endpoint returns a fixed body and performs no I/O whatsoever. In the 32-connection run it reaches 10,688 req/s on ONRouter and 10,777 req/s on OpenResty, a ratio of 0.99x. Two servers that differ by 1% on a no-work endpoint are not being favoured by the harness, which is what licenses us to attribute the other deltas to the implementations.

The comparison is not like-for-like in the process model. On Windows, nginx can only run a single worker using a `select()` event model, whereas ONRouter defaults to 16 Lua VMs and 18 worker threads on a shared work-stealing pool. The absolute numbers therefore compare *configurations available on the platform*, not two identical process models. A Linux comparison is planned and is not included here.

5.2 Throughput and latency

Figure 3 plots throughput and Figure 4 plots median latency. Table 2 gives the ratios.

5.3 Interpretation

The static path wins because of the cache, not the engine. The static endpoint serves a file of about 2.6 KB that fits inside the 1 MB in-memory cache threshold, so after the first request it is served from memory with a single `stat`. OpenResty opens the file on every request. This is a caching difference, not an engine difference, and the paper says so; the repository's own caveat list states the same.

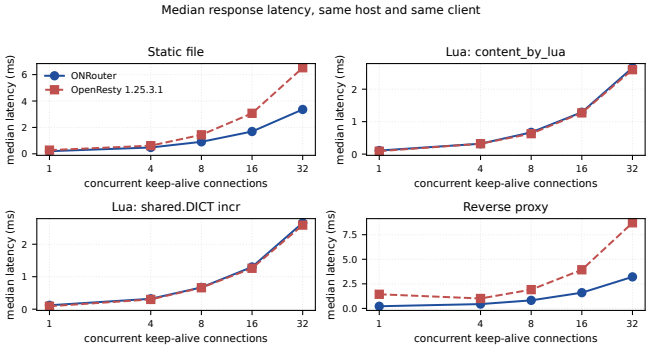


Figure 4: Median response latency at the same points.

Table 2: ONRouter throughput divided by OpenResty throughput, at each concurrency level. Values greater than one favour ONRouter. The `return 200` row is the no-I/O control.

Scenario	1	4	8	16	32
Control: `return 200`	—	—	—	—	0.99
Static file	1.27	1.78	1.79	1.82	1.78
Lua: `content_by_lua`	0.74	0.93	0.95	0.98	0.99
Lua: `shared.DICT`	0.83	0.93	1.00	0.97	0.99
Reverse proxy	26.58	2.34	2.28	2.44	2.49

The reverse proxy path wins because of the event model. The largest sustained ratio is on the reverse-proxy scenario, where the backend is a Python `BaseHTTP` server. At one connection OpenResty reaches only 140 req/s against ONRouter's 3,726 req/s, which is a single-connection `select()` artefact rather than a throughput result. At 16 and 32 connections, where the comparison is meaningful, the ratio settles at 2.44x and 2.49x, and the median latency gap is 1.60 ms against 3.93 ms and 3.21 ms against 8.71 ms. The async proxy is better at holding many slow backends concurrently.

The Lua paths lose, and the reason is the interpreter. At one connection ONRouter's Lua endpoints run at 0.74x and 0.83x of OpenResty's. This is Lua 5.4 interpreted against LuaJIT, exactly as the repository's gap analysis predicts, and the effect shrinks monotonically with concurrency — 0.93x at four, 0.99x at thirty-two — because by then the client, not the script, is the bottleneck. A user who writes a CPU-heavy Lua handler will get slower performance from ONRouter, and we state that plainly rather than averaging it away. The mitigating design decision is the VM pool: because each VM is independent and the selection is atomic, the interpreted runtime is not forced onto a single lock, which is what keeps the 8-connection number from collapsing.

Endpoint equivalence. The harness checks that the two servers produce the same status code and the same body length for each endpoint. Four of the five match exactly. The fifth, `content_by_lua`, differs by three bytes (57 against 54); the difference is in the body the script emits, not in the gateway's framing, and the two configurations are not the same

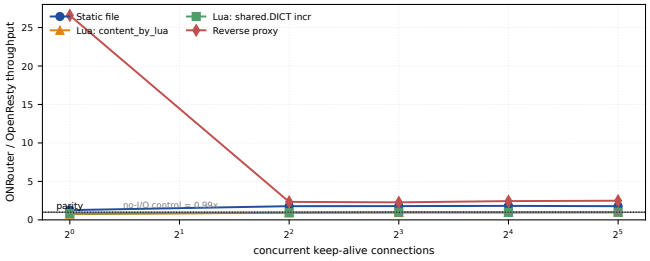


Figure 5: Throughput ratio against concurrency. The control endpoint sits on the parity line; the static and reverse-proxy paths separate from it; the Lua paths approach it from below as the client becomes the bottleneck.

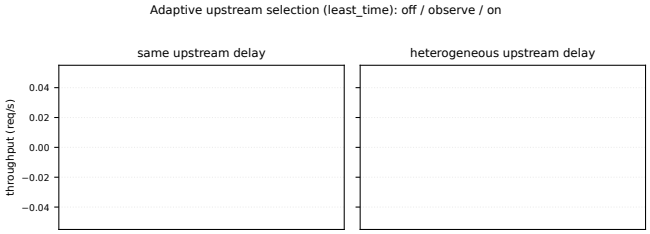


Figure 6: Adaptive upstream selection. In the homogeneous case the adaptive tiers cost slightly more than the fixed floor. In the heterogeneous case the `observe` tier identifies 1,525 of 3,000 decisions that would differ, and the `on` tier applies them.

script. We report it rather than tuning the endpoint until the numbers matched.

5.4 Adaptive upstream selection

Figure 6 shows the `least_time` tiers on a synthetic two-backend scenario. In the homogeneous case the adaptive tiers are marginally *slower* than the fixed floor, because the selection overhead is pure cost when every backend is identical. In the heterogeneous case — one backend delayed by 10 ms and another by 40 ms — the `observe` tier samples 3,000 decisions and finds that 1,525 of them would have differed from the floor's choice, while the `on` tier actually applies 3,000 of them with no cold start. The result shows the mechanism is doing what it claims; the throughput difference at this scale is inside the run-to-run spread.

5.5 Security evaluation

Three matrices in the repository test the security properties directly.

Protocol-level checks. `sec_test.py` contributes 30 checks covering path traversal (RFC 3986 Berners-Lee et al. (2005) dot-segment normalisation, with `../`, `..%2f`, `%252e`, backslash, and null-byte forms all rejected and an escaping path answered with 400), request smuggling (duplicate `Content-Length`, illegal `Content-Length`, and

`Content-Length` together with `Transfer-Encoding` all answered with 400, and a missing `Host` on HTTP/1.1 answered with 400), CRLF injection, malformed requests, oversized requests (501 for an unknown method, 414 for a URL above 8 KB, 431 for oversized headers, 413 for an oversized body rejected before the body is read), and the default response-header set. Slow connections are covered by the runtime model rather than by a special case: because no worker thread is dedicated to a connection between reads, the classic Slowloris pattern Hansen (2009) does not monopolise a thread, and streamed uploads (`proxy_request_buffering off`) are bounded by `client_body_timeout`, whose limit is per inter-read interval rather than total body time.

Real-traffic reproduction of a documented audit. `tools/security_matrix_e2e.py` contributes 36 checks that replay each finding from `docs/SECURITY_AUDIT.md` against a running gateway, one gateway process per probe. The probes are designed so that each one fails against the pre-fix binary; the value of the matrix is that it is tied to specific documented findings rather than to generic assertions.

Remote-execution surface. `tools/rce_matrix_e2e.py` contributes 27 checks against the paths where a gateway can be induced to execute attacker-controlled text: Lua-source and path injection, `internal` location escape, `proxy_pass` SSRF against a canary listener, and encoded-CRLF request splitting. The backend used by these probes echoes the exact request line it received, so a split is detected by observation rather than by a proxy's opinion.

Fail-closed access control. `auth_basic` without a user file denies with 401 rather than serving. htpasswd entries with an empty hash are dropped, so an empty password cannot log in. A peer address that cannot be parsed denies rather than being skipped, so a malformed address cannot bypass an `allow`/`deny` list. The htpasswd cache is bounded by a one-second staleness limit on top of `(mtime, size)` invalidation, so a revoked credential stops working even when the file is rebuilt with identical timestamps. Password comparison follows the constant-time discipline of Stinson (2006): entries are stored and compared as digests, and a stored value with an empty hash is rejected before any comparison is attempted.

Bounded blast radius. The `sub_filter` buffer is capped at 8 MiB per response and larger files stream unfiltered with a warning. Lua coroutines are preempted every `lua_slice_instructions` instructions, so a runaway loop degrades one VM rather than the process.

No front/back URI disagreement. The `Upgrade` (WebSocket) tunnel forwards the client's raw, undecoded request target, so the URI that routing and access control decide on is the same URI the backend parses.

5.6 Threats to validity

The client is the bottleneck. The load generator is Python. It saturates around 9,591 req/s on this host, and the server-side rates at 16 and 32 connections are close to that ceiling. This is the classic client-bound measurement, in which the observed rate is a property of the closed system rather than of the server Little (1961); Amdahl (1967). Absolute rates are therefore not portable and the high-concurrency ratios are compressed toward one. The cross-implementation comparison is meaningful because both servers face the same client on the same host, but it is not a capacity measurement.

The platform is not a like-for-like process model. As described above, nginx on Windows is restricted to one worker with a `select()` loop. The reverse-proxy ratio at one connection in particular is a platform artefact. The 16- and 32-connection figures, where the comparison is meaningful, are the ones we rely on.

The static comparison is a cache comparison. OpenResty was not configured with an equivalent in-process body cache. The static ratio measures ONRouter's cache against OpenResty's default behaviour, not two cache implementations against each other.

One host, one build. All measurements come from a single Windows host and a single build of each server. We do not have a Linux cross-check, and the repository's own `docs/LINUX_AB.md` records that comparison as pending.

No user study. We make no claim about operator productivity, configuration familiarity, or the quality of the scripting experience. We compare systems, not people.

6 Discussion

6.1 What the Rust rewrite bought

Three things, and they are not the ones usually claimed.

Preemption is possible at all. In a C gateway, a runaway Lua loop cannot be interrupted without a cooperative yield; the usual mitigations are a wall-clock timeout on the request and killing the worker, which loses every other request the worker was serving. Because the Lua VM here is behind an embeddable API whose instruction counter can be polled, the interruption can be placed at a point where the only casualty is the offending coroutine.

Blocking is a compile-time concern, not a convention. In C, a blocking call on the request path is invisible in review and fatal in production. In Rust, it is at least visible; in this codebase it was converted into an explicit list during the blocking analysis, and each entry was resolved individually.

Trust ordering became a property of the code path. The front-proxy rules are stated in the README, but the code makes them structural: the trust gate is consulted before the profile is, the promotion path requires an explicit opt-in, and the benchmark drives each mode with real traffic. This is the difference between a convention and a guarantee Adya (1999).

6.2 What it did not buy

Interpreter performance. Interpreted Lua 5.4 does not match LuaJIT, and the measurements are consistent with that. The paper does not argue otherwise.

Process model maturity. ONRouter is a single process. It has no master/worker split, no binary upgrade path, and no `USR2` support. The design for the latter exists as a document (`docs/USR2_HOT_UPGRADE.md`) and a test plan, not as code. For deployment patterns that depend on a zero-downtime binary swap, this is a real gap.

Ecosystem breadth. OpenResty's advantage is not only the engine but the accumulated library: `resty.websocket`, `resty.jwt`, `resty.mysql`, `resty.dns`, `resty.upload`, OCSP stapling, session stores. ONRouter implements a substantial `ngx.*` surface — timers, TCP and UDP cosockets, light threads, semaphores, subrequests including concurrent `capture_multi`, shared dictionaries, `resty.lock`, `resty.http`, `resty.redis`, `resty.string`, `resty.cookie`, `resty.limit.req|count|conn` — but the long-tail semantics of individual functions are not identical, and the gap analysis tracks the differences row by row.

Operator familiarity. A gateway that re-implements nginx's configuration dialect is only nginx-compatible to the extent that the two implementations agree. The compatibility scripts in `tools/` pin a specific subset; where they and nginx disagree, ONRouter follows nginx.

7 Conclusion

Re-implementing the OpenResty programming model in Rust is not primarily a performance exercise. The interesting outcome is that a memory-safe implementation forces a set of questions that a C implementation tends to leave implicit: where does blocking I/O happen, what happens to a script that never yields, and in what order does the gateway learn who the client is. ONRouter's answers — an explicit per-path blocking analysis, a preemptible Lua runtime with a non-blocking fast path, and a front-proxy seam placed after the trust gate — are the substance of this paper.

The measurements are mixed and we have kept them that way. ONRouter is 1.8x faster on the static path and 2.5x faster on the reverse-proxy path at meaningful concurrency, 0.74x slower on CPU-bound Lua at a single connection and at parity by thirty-two, and indistinguishable on a no-work control endpoint. A gateway that is faster on I/O-bound paths, no slower once the client saturates, and interruptible in a way its C counterpart is not, is a defensible alternative — and the repository ships the harness that lets a reader check the claim rather than take it on faith.

References

hyper — a fast and correct http/1 and http/2 implementation for rust, 2026. https://hyper.rs/.

A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

T. Berners-Lee, R. Fielding, and L. Masinter. Rfc 3986: Uniform resource identifier (uri): Generic syntax. Technical report, Internet Engineering Task Force, 2005.

G. Einziger, R. Friedman, and B. Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage*, 13(4):1–31, 2017.

R. Fielding, M. Nottingham, and J. Reschke. Rfc 9110: Http semantics. Technical report, Internet Engineering Task Force, 2022a.

R. Fielding, M. Nottingham, and J. Reschke. Rfc 9112: Http/1.1. Technical report, Internet Engineering Task Force, 2022b.

R. Hansen. Slowloris — http denial of service attack, 2009. https://ha.ckers.org/slowloris/.

R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. *Lua 5.4 Reference Manual*. Lua.org, 2020. Debug library and the instruction-count hook used for preemption.

R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2 (POPL):Art. 66, 2018.

S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2019.

C. Lerche. Tokio — an asynchronous runtime for rust, 2026. https://tokio.rs/.

J. D. C. Little. A proof for the queuing formula: $l = \lambda w$. *Operations Research*, 9(3):383–387, 1961.

MITRE. Cwe-444: Http request/response smuggling attack, 2026a. https://cwe.mitre.org/data/definitions/444.html.

MITRE. Cwe-22: Improper limitation of a pathname to a restricted directory ('path traversal'), 2026b. https://cwe.mitre.org/data/definitions/22.html.

M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

mlua-rs. mlua — rust bindings to lua, 2026. https://github.com/mlua-rs/mlua.

nginx. nginx documentation: directives, modules, and the request phase model. Technical report, nginx.org, 2026. https://nginx.org/en/docs/.

M. Nottingham. Rfc 9211: The cache-status http response header field. Technical report, Internet Engineering Task Force, 2022.

OpenResty. lua-nginx-module — embed lua into the nginx http and websocket server, 2026. https://github.com/openresty/lua-nginx-module.

M. Pall. Luajit — a just-in-time compiler for lua, 2026. https://luajit.org/.

D. R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 3rd edition, 2006.

P. Viotti and M. Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys*, 49(1): 1–34, 2016.

Y. Zhang. Openresty — a dynamic web platform based on nginx and luajit, 2014. https://openresty.org/.