loadgen-rs: An HTTP Benchmark Client in Rust

2026-03-03

How I built an HTTP benchmark client in Rust that supports HTTP/1.1, HTTP/2, and HTTP/3 — machine-readable output, distributed multi-node mode, Terraform/Ansible cloud deployment, and what CPU profiling revealed about the performance ceiling.

TL;DR / Executive Summary

Prefer a shorter read first?
loadgen-rs (Short): Results, Quick Start, and Tool Choice

loadgen-rs is a Rust HTTP benchmark client that covers HTTP/1.1, HTTP/2, and HTTP/3 in one tool, outputs machine-readable JSONL/CSV, and scales from single-node runs to distributed multi-worker tests.

- H1/H2 throughput is competitive with h2load, while H3 still favors h2load in peak throughput.
- loadgen-rs is strong when you want automation-friendly output and one consistent workflow across protocols.
- Distributed mode is a first-class feature, not an afterthought: controller + workers + correct histogram merge.
- A Deno/FFI layer enables scripted, k6-style scenarios with checks, extractors, and correlation.

What Is loadgen-rs?

Why I Built It

If you benchmark HTTP servers, you probably know h2load from the nghttp2 project. It's a very solid and mature tool, but there are a few pain points:

- **HTTP/3 support is not enabled by default** — h2load can do H3/QUIC, but only via a separate build variant with ngtcp2.
- **Requires a C compiler and many native libraries** — this makes source builds and CI setup more cumbersome, especially across different environments.

I wanted a tool that tests all three HTTP protocols and outputs the results directly as JSON and *easy* to build. So I built **loadgen-rs**.

Core Capabilities (H1/H2/H3, JSONL, Distributed, Scripting)

Component	Crate	Why
Runtime	tokio (multi-thread)	De-facto standard for async Rust
HTTP/1.1	Raw TCP + httparse	Maximum control, zero framework overhead
HTTP/2	h2 (direct, no hyper)	Avoids hyper's abstraction layer
TLS	rustls (ring backend)	Pure Rust, no OpenSSL dependency hell
HTTP/3	quinn + h3 + h3-quinn	QUIC stack in Rust
CLI	clap (derive)	Ergonomic, type-safe
Metrics	hdrhistogram	Industry standard for latency measurements
Allocator	mimalloc	Better multi-threaded allocation performance

An early design choice was to bypass hyper entirely. For H1, we use raw TCP sockets with `httparse` for response parsing. For H2, we use the `h2` crate directly. This eliminates one abstraction layer and gives full control over connection management, flow control tuning, and request templating. Hyper is a great library, but its abstraction model didn't fit the needs of this benchmarking tool.

In addition to single-node CLI benchmarking, loadgen-rs includes distributed controller/worker execution and scripted scenario support through a native FFI layer for Deno.

Runtime and Execution Model

Figure 1: Runtime and execution model: main, runner, drivers, metrics, output

Connection Strategy

The `-c` and `-m` flags follow h2load semantics:

- **`-c N`** spawns N parallel connections distributed across worker threads
- **`-m N`** limits concurrent in-flight requests per connection (H2/H3: concurrent streams; H1: in-flight cap)
- Total parallelism: `c * m`

Each worker thread runs its own `current_thread` Tokio runtime — **no cross-thread synchronization in the hot path**. Workers own local HdrHistograms that are merged only at the end.

Count vs. Duration Mode

- **Count mode** (`-n 1000`): A shared `AtomicU64` budget that workers decrement via CAS
- **Duration mode** (`-D 10s`): A `CancellationToken` that fires after the specified duration

- `-D` takes priority over `-n` (matching h2load behavior)

RPS Semantics (h2load-compatible)

`--rps` is implemented as a **per-client** request start rate, matching `h2load`.

- Per-client target: `--rps`
- Aggregate target: `--rps * -c`

This matters for comparisons: - `-c 10 --rps 20` targets roughly `200` started req/s overall - `-c 10 --rps 200` targets roughly `2000` started req/s overall

Also note: in short **count mode** runs (`-n`), achieved `%` can appear above or below 100% because startup and shutdown overhead dominate a short elapsed window. For stable pacing comparisons, use **duration mode**.

Output and TLS Notes

The JSONL output contains everything a Python script needs:

```
{
  "proto": "h2",
  "url": "https://target:8443/",
  "clients": 8,
  "threads": 4,
  "rps": 12543.7,
  "latency_p50_us": 320,
  "latency_p90_us": 890,
  "latency_p99_us": 2100,
  "status_counts": {"200": 125437},
  "status_2xx": 125437,
  "err_total": 0,
  "elapsed_s": 10.001
}
```

In Python:

```
import json
results = [json.loads(line) for line in open("results.jsonl")]
df = pd.DataFrame(results)
df.plot(x="clients", y="rps", kind="bar")
```

TLS Configuration

`--insecure` implements a custom `ServerCertVerifier` that accepts everything. `--tls-ciphers` allows selecting specific cipher suites — with validation that H3 only permits TLS 1.3 ciphers. Custom CA certificates can be loaded via `--tls-ca`.

Quick Start

Single-Node Benchmark in 30 Seconds (cli)

```
# HTTP/1.1, 1000 requests
loadgen-rs -n 1000 -c 4 -t 2 --h1 'http://bench.local:8081/?s=256k'

# HTTPS
loadgen-rs -n 1000 -c 4 -t 2 --h1 --insecure 'https://bench.local:8082/?
s=256k'

# HTTP/2, 10 seconds
loadgen-rs -n 0 -D 10s -c 8 -t 4 -m 10 \
  --alpn-list=h2 --insecure 'https://bench.local:8082/?s=256k'

# HTTP/3 (QUIC)
loadgen-rs -n 0 -D 10s -c 4 -t 2 \
  --alpn-list=h3 --insecure 'https://bench.local:8082/?s=256k'
```

Single-Node Benchmark in 30 Seconds (container)

If you don't have `podman`, you can use `docker` instead.

```
# HTTP/3 (QUIC) 10 seconds
podman run -it --rm \
  --network host \
  --name gh-loadgen ghcr.io/git001/loadgen-rs:v0.3.0 \
  -D 10s -c 8 -t 4 -m 10 \
  --alpn-list=h3 \
  --insecure \
  'https://bench.local:8082/?s=256k'
```

Distributed Benchmark in 2 Minutes

```
# Start workers (one per machine, or locally for testing)
bash examples/worker-start.sh

# Run controller
deno run --allow-ffi --allow-net --allow-read --allow-env \
  examples/distributed.ts http://worker1:9091 http://worker2:9091
```

Or programmatically:

```
import { Controller } from "./ts/mod.ts";

const controller = new Controller({
  workers: ["http://worker1:9091", "http://worker2:9091"],
  config: {
```

```
    url: "https://target:8443/",
    protocol: "h2",
    clients: 200,     // split: 100 per worker
    threads: 4,
    duration_s: 30,
    insecure: true,
  },
});

const result = await controller.run();
console.log(result.merged_report.rps);
```

Minimal Comparison Against h2load

```
# 1) H1 count mode (same request budget)
h2load --h1 -n 200 -c 10 -t 8 -m 1 "https://bench.local:8082/?s=256k"
loadgen-rs --h1 -n 200 -c 10 -t 8 -m 1 "https://bench.local:8082/?s=256k"

# 2) H1 duration mode with pacing (~200 aggregate rps: 10 clients * 20
rps/client)
h2load --h1 -D 10s -c 10 -t 8 -m 1 --rps 20 "https://bench.local:8082/?s=
256k"
loadgen-rs --h1 --duration 10s -n 0 -c 10 -t 8 -m 1 --rps 20 "https://
bench.local:8082/?s=256k"

# 3) H2 duration mode with pacing (same effective target)
h2load --alpn-list=h2 -D 10s -c 10 -t 8 -m 1 --rps 20 "https://bench.
local:8082/?s=256k"
loadgen-rs --alpn-list h2 --duration 10s -n 0 -c 10 -t 8 -m 1 --rps 20
"https://bench.local:8082/?s=256k"

# 4) H3 duration mode with pacing (same effective target)
h2load --alpn-list=h3 -D 10s -c 10 -t 8 -m 1 --rps 20 "https://bench.
local:8082/?s=256k"
loadgen-rs --alpn-list h3 --duration 10s -n 0 -c 10 -t 8 -m 1 --rps 20
"https://bench.local:8082/?s=256k"
```

Key Results vs h2load

After implementing and tuning the tool, I ran systematic A/B comparisons against `h2load` on the same hardware and target server.

High-Concurrency Benchmark

Setup: - Duration: 30 seconds per run, 30-second cooldown between runs - Configuration: `-c 512 -t 8 -m 8` - Target: `https://127.0.0.1:8082/?s=256k` (256 KB response body) - h2load runs via `podman run --rm --network host localhost/h2load`

Protocol	h2load RPS	loadgen-rs RPS	Ratio	h2load Latency (mean)	loadgen-rs Latency (mean)
H1	18,851	**19,148**	**101.6%**	27.07 ms	25.69 ms
H2	17,837	**18,025**	**101.1%**	125.59 ms	223.91 ms
H3	**9,006**	6,702	74.4%	450.15 ms	604.17 ms

Key observations:

- **H1**: loadgen-rs is **faster** than h2load on both throughput and latency. The raw TCP + httparse approach with pre-built request templates pays off.
- **H2**: Throughput is essentially identical. However, mean latency is ~1.8x higher in loadgen-rs due to an architectural difference (see profiling section below).
- **H3**: h2load (using ngtcp2/nghttp3 in C) retains a ~25% throughput advantage over quinn/h3 in Rust.

To be fair, h2load was sometimes better than loadgen-rs, and vice versa.

Resource Usage (CPU and Memory)

For resource fairness, both tools were run as containers and sampled with `podman stats`:

Setup: - 3 repeats per point, `duration=5s`, `c=4`, `t=2`, `m in {1,8}`

Metric	H1/H2	H3
CPU time	mostly `+2%` to `+8%`	`-1%` to `+10%`
Peak memory	about `+410%` to `+914%`	about `+1393%`

In absolute terms, peak memory was typically: - `h2load`: around `2.3 MB` to `3.0 MB` - `loadgen-rs`: around `15 MB` to `39 MB`

The higher memory usage comes from Tokio's runtime overhead, per-worker HdrHistograms, and Rust's buffered I/O layers. In practice, even 39 MB is negligible for a benchmark client.

Which Tool Should You Choose?

Decision Matrix by Use Case

Priority	Recommendation
H1/H2 throughput + automation	**loadgen-rs** — matches or beats h2load, with native JSONL output
H3 peak throughput	**h2load** — ngtcp2 is ~25% faster than quinn
Lowest memory footprint	**h2load** — 3 MB vs 15-39 MB
Cross-protocol consistency	**loadgen-rs** — same tool, same output format for H1/H2/H3
Multi-machine load generation	**loadgen-rs** — built-in distributed mode with correct histogram merge
Cloud-provisioned benchmarks	**loadgen-rs** — Terraform + Ansible pipeline for Hetzner Cloud
Distributed scripted scenarios	**loadgen-rs** — workers run multi-step checks/extractors via `POST /run-script`

Practical Recommendation

For most benchmarking workflows, the choice comes down to this: do you need the absolute highest H3 throughput, or do you want a single binary with all protocols available out of the box?

Distributed Mode Overview

A single loadgen-rs process can saturate most test targets, but sometimes you need more — testing a load balancer with geographically distributed traffic, generating more requests than one machine's NIC can handle, or simply simulating a more realistic multi-source access pattern.

Architecture

The distributed mode uses a TypeScript controller (running in Deno) that orchestrates multiple worker-agents over plain HTTP/JSON — no gRPC or custom protocol needed:

Distributed architecture: controller and workers with report merge

Figure 2: Distributed architecture: controller and workers with report merge

Each worker is a lightweight Deno HTTP server (~120 LOC) that wraps the same `LoadgenFFI` batch API used for single-node benchmarks. The controller splits the total client count (`-c`) evenly across workers, sets a coordinated start timestamp, and fires all jobs concurrently.

The histogram merge problem

Merging latency results from distributed workers is deceptively tricky. The naive approach — averaging p99 values from each worker — is **statistically wrong**. If worker A has p99 = 5ms (from 50K requests) and worker B has p99 = 15ms (from 50K requests), the combined p99 is *not* 10ms. It depends on the full distribution shape.

The correct approach is to merge the underlying histograms. Each worker serializes its HdrHistogram using V2-Deflate format (a compact binary encoding, typically 1–5 KB per histogram), base64-encodes it, and includes it in the `RunReport` JSON:

```
{
  "rps": 136949.0,
  "latency_p99_us": 210,
  "latency_hist_b64": "HISTFAAAB...",
  "ttfb_hist_b64": "HISTFAAAB...",
  "connect_hist_b64": "HISTFAAAB..."
}
```

The controller passes all worker reports to `loadgen_merge_reports()` — a new FFI export that deserializes each histogram, merges them with `Histogram::add()`, and recomputes all percentile fields from the combined distribution. This is the same merge algorithm used internally for per-thread metrics within a single process.

Coordinated start

For benchmark results to be meaningful, all workers should start at the same time. The controller calculates `start_at = now + 500ms`, includes it in each worker's `/run` request, and each worker sleeps until that timestamp before starting the benchmark. Clock skew under 10ms is acceptable

for benchmark purposes — and on a local network or the same machine, skew is typically sub-millisecond.

Minimal new code, maximum reuse

The implementation reuses the existing FFI infrastructure almost entirely:

Component	New code	Reused
Rust: `merge_distributed_reports()`	~120 LOC	Existing `merge_metrics()` pattern
Rust: FFI export	~25 LOC	Existing `catch_unwind` + error slot pattern
TypeScript: worker-agent	~120 LOC	`LoadgenFFI` class
TypeScript: Controller	~150 LOC	`mergeReports()` FFI wrapper

Only two new Rust dependencies were needed: `base64` (for histogram serialization) was already an indirect dependency via reqwest. The `export_histograms` field in `BenchConfig` defaults to `false`, so the existing CLI and single-node FFI path are completely unaffected.

Real-world validation

Testing 2 local workers against the same target (H2, 8 total clients, 5s):

Setup	RPS	p99 Latency
Single-node (`-c 8 -m 4`)	284,226	185 us
Distributed (2 × 4 clients)	269,351	207 us
Ratio	**94.8%**	

The ~5% overhead comes from the HTTP/JSON coordination round-trip and the coordinated start delay. In a real multi-machine deployment (where each worker has its own NIC and CPU), total throughput scales linearly — you're no longer bound by a single machine's resources.

Scripted Scenarios Overview (Deno / k6-style)

After the initial release, I added a TypeScript/Deno layer on top of the native FFI to make scripted testing and k6-style output easier to compare.

What was added

- A typed Deno SDK in `ts/`:
  - `LoadgenFFI` for run-level benchmarking
  - `LoadgenStepFFI` for step/session execution
- A reusable `printK6LikeSummary()` formatter used by the Deno examples
- New comparison examples:
  - `examples/ab-compare.ts` and `examples/k6/ab-compare.ts` (closed model)
  - `examples/ab-compare-rps.ts` and `examples/k6/ab-compare-rps.ts` (fixed RPS)
- A one-command comparison helper:
  - `scripts/ab_compare_report.sh` (prints req/s, avg, p99, fail rate, H2 check)

Important fairness fix

One easy mistake is RPS interpretation:

- In loadgen-rs, `--rps` is **per client**
- Aggregate target is `rps_per_client * clients`

The Deno comparison script now maps a total target RPS to per-client RPS automatically (`RPS / CLIENTS`) to match k6 constant-arrival tests.

Why latency can differ even at similar req/s

Even when request rate is aligned, latency can diverge if concurrency shape is not aligned:

- number of active clients/VUs
- streams per connection (`-m` / `max_streams`)
- thread count and connection reuse behavior

In short: equal target RPS does not automatically imply equal queueing behavior.

Stage profile support (k6-like)

I also added `examples/ab-stages.ts`, which emulates stage ramps by splitting stages into short segments and running sequential FFI runs. This gives a practical equivalent for basic k6 stage workflows.

Correlation path

For dynamic data and sequence scenarios, `runScriptScenario()` supports:

- `execution_mode: "fetch"` (pure Deno)
- `execution_mode: "ffi-step"` (native step transport)

The step API design and contract are documented in: - `docs/script-mode-ffi.md` - `docs/ffi-step-api.md` - `docs/ffi-contract.md`

k6 Parity Snapshot (as of March 1, 2026)

Short answer: Deno + loadgen-rs is now close to k6 for HTTP-focused scripted workflows, but it is not yet a full 1:1 replacement.

What is now very similar:

- Scripted scenarios with step chains, checks, and extractors
- Correlation and dynamic data (`{{var}}` templates, JSON/regex/header/DOM extract)
- Cookie handling and redirect policies
- Two execution paths: pure `fetch` and native `ffi-step`
- Multi-protocol support in native step mode (`h1`, `h2`, `h3`)
- k6-like terminal summary output and side-by-side comparison scripts

Where k6 is still stronger:

- More mature executor/scenario system overall (arrival/vu/stage/ramping breadth)
- Larger built-in ecosystem (`k6/http`, ws/browser modules, extension landscape)
- Richer output/integration stack (cloud and common observability backends)
- Longer production track record for large shared test suites and CI pipelines

Practical takeaway:

If your priority is HTTP load + correlation + native H1/H2/H3 coverage with comparable CLI summaries, Deno + loadgen-rs is already highly usable. If you need the full k6 orchestration and ecosystem depth, k6 still has an edge.

The distributed mode isn't limited to raw throughput benchmarks. Workers also support `POST /run-script` — the same scripted scenario engine used locally, but executed remotely on each worker. This means you can run multi-step request sequences with checks, extractors, cookies, and response validation across a fleet of machines.

The implementation required minimal changes: the worker-agent imports `runScriptScenario()` from the existing `script_mode.ts` and exposes it as a new HTTP endpoint. The controller sends a `ScriptScenarioConfig` (same type used for local script mode) to each worker and collects `ScriptRunResult` objects with per-check pass/fail counters.

A practical example ships as `examples/html-check.ts` — a verification script that tests whether each worker can reach a target URL and parse the response correctly:

```
const config: ScriptScenarioConfig = {
  vus: 3,
  duration_s: 5,
  execution_mode: "ffi-step",
  step_session_config: {
    protocol: "h2",
    insecure: true,
    response_headers: true,
    response_body_limit: 2_000_000,
  },
  steps: [{
    name: "html-get",
    method: "GET",
    url: targetUrl,
    capture_body: true,
    checks: {
      status_in: [200, 403],
      body_includes: ["<title>Example</title>"],
      header_includes: { "set-cookie": "session=" },
    },
  }],
};
```

The controller sends this config to each worker via `POST /run-script`, then aggregates the results into a per-worker table. If the target returns both the HTML element and the cookie on every single response, the iteration count, HTML match count, and header match count should be identical — any mismatch indicates a parsing or check evaluation bug:

```
=== per-worker results ===

  worker                                    iterations  html_match
header_match  ok?
------------------------------------------------------------------------------------------
  http://worker1:9091                          62          62
62  PASS
  http://worker2:9091                          47          47
47  PASS
------------------------------------------------------------------------------------------
  TOTAL                                       109         109
109  PASS
```

All counts match perfectly across both workers. This validates the full chain: Terraform-provisioned infrastructure, Ansible-deployed workers, `ffi-step` execution mode with reqwest on the Rust side, response header capture, and the check evaluation engine — all working end-to-end on remote machines.

The `header_includes` check type was added specifically for this use case. Unlike `header_exists` (which only checks presence) or `header_equals` (which requires an exact match), `header_includes` tests whether a header value *contains* a substring — essential for `Set-Cookie` headers where the full value includes expiry, path, and other attributes beyond the cookie name.

Multi-Step Login/Logout Workflow

The scripted scenario engine really shines when steps depend on each other. A second distributed example — `examples/roundcube-login.ts` — verifies a full Roundcube Webmail login/logout cycle across workers. Unlike the single-step HTML check, this scenario chains three HTTP requests with state flowing between them:

```
steps: [
  {
    name: "login-page",
    method: "GET",
    url: "https://mail.example.com/mail/",
    capture_body: true,
    checks: {
      status_in: [200],
      body_includes: ["Roundcube Webmail :: Welcome to Roundcube
Webmail"],
    },
    extract: [{
      type: "dom",
      selector: 'input[name="_token"]',
      attribute: "value",
      as: "token",
```

```
    }],
  },
  {
    name: "login-submit",
    method: "POST",
    url: "https://mail.example.com/mail/?_task=login",
    headers: { "content-type": "application/x-www-form-urlencoded" },
    body: "_task=login&_action=login&_timezone=...&_token={{token}}
&_user=...&_pass=...",
    capture_body: true,
    checks: {
      status_in: [200],
      body_includes: ["Roundcube Webmail :: Inbox", 'data-label-msg="The
list is empty."'],
    },
    extract: [{
      type: "regex",
      pattern: '"request_token":"([^"]+)"',
      group: 1,
      as: "logout_token",
    }],
  },
  {
    name: "logout",
    method: "GET",
    url: "https://mail.example.com/mail/?_task=logout&_
token={{logout_token}}",
    capture_body: true,
    checks: {
      status_in: [200],
      body_includes: ["Roundcube Webmail :: Welcome to Roundcube
Webmail"],
    },
  },
]
```

Each step depends on extracted values from the previous one: `{{token}}` (the CSRF token extracted from the login form's hidden input via a DOM CSS selector — `input[name="_token"]`) flows into the POST body, and `{{logout_token}}` (the `request_token` extracted from the Roundcube JavaScript via regex) flows into the logout URL. The `dom` extractor parses the HTML response with deno-dom's `DOMParser` and runs `querySelector` — far more robust than regex for structured HTML. Session cookies are managed automatically by the native `cookie_jar` — the `roundcube_sessid` and `roundcube_sessauth` cookies persist across all three steps within an iteration, and Roundcube's logout response invalidates the session so the next iteration starts clean.

```
=== per-worker results ===

  worker                                    iterations    login_ok
logout_ok  ok?
------------------------------------------------------------------------------
  http://worker1:9091                        45          45          45
PASS
  http://worker2:9091                        45          45          45
PASS
------------------------------------------------------------------------------
  TOTAL                                      90          90          90
PASS
```

All 90 iterations completed successfully across both workers: every login extracted a valid CSRF token, every POST received the inbox page, and every logout returned to the login screen. This validates the full extractor → template substitution → cookie management pipeline running inside the Rust FFI layer on remote machines.

Deep Dive: Performance Analysis

To understand the performance gaps, I profiled loadgen-rs using `perf` and samply. The setup:

```
# Build with debug symbols but full optimization
cargo build --profile profiling  # inherits release, adds debug=2

# Record CPU profile
perf record -g --call-graph dwarf -o /tmp/perf-h2.data -- \
  ./target/profiling/loadgen-rs --alpn-list h2 -D 10s -c 64 -t 4 -m 8 \
  --insecure https://127.0.0.1:8082/?s=256k
```

H2 CPU Profile

Self %	Function	Category
27%	`ring::aes_gcm_dec_update`	TLS decrypt (AES-256-GCM)
21%	`__memmove_avx` (libc)	Memory copies
16%	Kernel	TCP read/write syscalls
2.0%	`rustls::read_opaque_message_header`	TLS framing
1.4%	`_mi_page_malloc`	Allocator (mimalloc)
1.3%	`h2::Connection::poll2`	H2 protocol state machine
1.3%	`h2::DynConnection::recv_frame`	H2 frame processing
1.2%	`BytesMut::split_to`	Buffer management

~64% of CPU time is spent outside of our code entirely — in TLS decryption, memory copies, and kernel syscalls. Only ~2.5% is in the h2 crate's protocol logic.

The H2 latency gap (1.8x vs h2load) is *not* a CPU bottleneck — it's architectural. The h2 crate uses an async mpsc channel between `SendRequest` (our task) and the connection driver task (h2's internal

I/O loop). Each request crosses this channel boundary at least twice (send + receive), adding idle time that doesn't show up in CPU profiles.

h2load, by contrast, runs everything in a single libev event loop tick — nghttp2 processes the request, writes the frame, and reads the response without any task switching.

H3 CPU Profile

Self %	Function	Category
15%	`ring::aes_gcm_dec_update`	TLS decrypt
13%	`__memmove_avx` (libc)	Memory copies
12%	Kernel	UDP recvmsg syscalls
3.5%	`quinn_proto::Connection::process_payload`	QUIC packet processing
3.1%	`quinn_proto::Connection::handle_decode`	QUIC packet decode
2.9%	`quinn::endpoint::RecvState::poll_socket`	UDP socket I/O
2.8%	`binary_heap::PeekMut::pop`	Timer/scheduling (loss detection, CC)
1.8%	`ring::aes_gcm::open`	Additional TLS overhead
1.7%	`quinn::RecvStream::poll_read_chunk`	Stream receive
1.7%	`quinn_proto::Endpoint::handle`	Packet routing/demux
1.6%	`h3::proto::varint::VarInt::decode`	H3 framing

The QUIC protocol stack (quinn-proto) accounts for **~15% of CPU** — compared to ~2.5% for h2. This is the fundamental cost of implementing QUIC in userspace Rust:

- **Per-packet crypto**: QUIC encrypts each UDP datagram individually (vs TLS record batching over TCP)
- **Congestion control state machine**: Quinn maintains timer heaps (`binary_heap::pop` at 2.8%) for loss detection and CC that TCP offloads to the kernel
- **Packet demux**: `Endpoint::handle` routes each incoming UDP datagram to the correct connection — work that the kernel's TCP stack does for free

h2load uses ngtcp2 (C) and nghttp3 (C) which have tighter inner loops, less allocation overhead, and direct `writev()` for packet batching.

What the Profile Means

The profiling data tells a clear story: **we've hit the performance ceiling of the Rust crate ecosystem**, not a bug in our code.

For H2, the bottleneck is TLS + memcpy + kernel — things we can't optimize without replacing ring/rustls. For H3, quinn's QUIC protocol processing adds ~15% overhead that ngtcp2 avoids by being a leaner C implementation.

Optimization Attempts (What Worked and What Didn't)

During development, I tried several optimizations to close the gap with h2load:

What Worked

Optimization	Effect	Protocol
Raw h2 crate instead of hyper	Eliminated one abstraction layer	H2
Raw TCP + httparse instead of hyper for H1	+15-25% RPS vs hyper-based H1	H1
`initial_rtt(1ms)` for quinn	Faster congestion control ramp-up on low-latency links	H3
1 GB stream/connection windows	Matches h2load's window sizes	H2, H3
H2 flow control: 512 KB batch release	Reduces WINDOW_UPDATE frame overhead	H2
UDP socket buffers 4 MB via socket2	Fewer kernel drops under load	H3
mimalloc global allocator	Better multi-threaded malloc performance	All
Pre-built request templates	Avoid re-building HTTP requests per iteration	All
Per-worker current-thread runtimes	No cross-thread lock contention	All

What Didn't Work

Optimization	Expected	Actual	Why
Thread-local shared quinn::Endpoint	Less per-connection overhead	**-12% RPS**	UDP socket contention — all connections on one worker competed for the same socket
H2 immediate flow control release (per-chunk)	Lower latency	**-5.6% RPS**	Generated too many WINDOW_UPDATE frames and cross-task messages
Cubic congestion controller (instead of NewReno)	Match h2load's ngtcp2 CC	**-7% RPS**	Quinn's Cubic implementation doesn't scale well with 512 concurrent connections on localhost
http2 crate (0x676e67 fork with parking_lot)	Faster mutex in h2 hot path	**-2.3% RPS**	The h2 protocol logic is only 2.5% of CPU — faster locks don't help when locks aren't the bottleneck

The failed optimizations were valuable lessons: profiling *before* optimizing would have saved time. The http2 fork experiment, for instance, was motivated by the assumption that h2's internal locking was a bottleneck — but perf showed it was only 1.3% of CPU time.

Architectural Comparison: loadgen-rs vs h2load

Understanding *why* the gaps exist requires comparing the architectures:

H2: Async Tasks vs Single Event Loop

```
loadgen-rs:  Worker Task ──mpsc──▶ h2 Connection Driver Task ──▶ TCP
h2load:      libev event loop ──▶ nghttp2 ──▶ writev() ──▶ TCP
```

The h2 crate splits request handling across two async tasks connected by an mpsc channel. Each request crosses this boundary twice. h2load runs everything synchronously in one event loop tick — zero context switches per request.

This explains the latency gap (1.8x) without a throughput gap: the pipeline stays full, but each individual request takes longer to complete.

H3: Userspace QUIC vs Lean C Implementation

```
loadgen-rs:  quinn (Rust) ──▶ ring (Rust/asm) ──▶ UDP socket
h2load:      ngtcp2 (C) ──▶ OpenSSL/BoringSSL (C/asm) ──▶ UDP socket
```

Both implement the same QUIC protocol, but ngtcp2 has advantages: - Tighter C inner loops with less allocation - Direct `sendmsg`/`recvmsg` batching - Fewer abstraction layers between protocol logic and I/O

Quinn adds Tokio task scheduling, channel-based I/O between endpoint and connection state, and Rust's safety overhead (bounds checks, reference counting on `Bytes`).

Deep Dive: Internal Architecture

Enum-Based Driver Dispatch

Instead of `dyn Trait` with async boxing, I use an enum:

```rust
pub enum Connection {
    H1Raw(h1_raw::H1RawConnection),
    H2(h2::H2Connection),
    H3(h3::H3Connection),
}

impl Connection {
    pub async fn send_request(&mut self, config: &RequestConfig)
        -> Result<RequestResult, RequestError>
    {
        match self {
            Connection::H1Raw(c) => c.send_request(config).await,
            Connection::H2(c) => c.send_request(config).await,
            Connection::H3(c) => c.send_request(config).await,
        }
    }
}
```

Benefits: - No heap allocation per dispatch - Compiler can inline - No `Pin<Box<dyn Future>>` overhead

FFI Two-API Design (Batch vs Step)

The Deno integration isn't just a thin wrapper — it's built on a Cargo workspace with a dedicated `loadgen-ffi` crate that compiles to a 6.4 MB shared library (`libloadgen_ffi.so`) exposing 13 C ABI functions via `Deno.dlopen()`.

Two distinct FFI APIs

The shared library provides two API levels, each serving a different use case:

1. Batch API — for high-throughput load generation:

```
loadgen_create(config_json) → handle
loadgen_run(handle)         → report_json    (nonblocking in Deno)
loadgen_metrics_snapshot(handle) → json
loadgen_destroy(handle)
```

This wraps the existing `run_from_config()` engine — the same optimized H1/H2/H3 drivers, per-worker runtimes, and HdrHistograms that match h2load performance. A single `loadgen_run` call spawns the full benchmark and returns a complete `RunReport` as JSON.

2. Step API — for scripted request sequences with correlation:

```
loadgen_step_session_create(session_config_json) → session_handle
loadgen_step_execute(session_handle, request_json) → response_json
(nonblocking)
loadgen_step_session_reset(session_handle)
loadgen_step_session_destroy(session_handle)
```

The Step API uses **reqwest** as its HTTP client rather than the custom drivers. This is a deliberate architectural choice: reqwest provides built-in cookie jars, redirect following, H1/H2/H3 support, and connection pooling — features that the raw benchmark drivers intentionally omit for performance. In script mode, correctness (cookies, redirects, body capture) matters more than raw throughput.

Panic-safe FFI boundary

Every exported function is wrapped in `std::panic::catch_unwind()` to prevent Rust panics from unwinding across the C ABI boundary (which would be undefined behavior). Errors are stored in a thread-safe slot (`OnceLock<Mutex<Option<String>>>`) and retrieved via `loadgen_last_error()`. All returned `char*` strings must be freed with `loadgen_free_string()` — the contract is documented in `docs/ffi-contract.md`.

TypeScript script engine

On the Deno side, `runScriptScenario()` in `ts/script_mode.ts` implements a k6-inspired scenario runner with:

- **VU parallelism**: N virtual users run as concurrent `Promise.all` tasks
- **Duration-based execution**: VUs loop until the target duration expires
- **Two execution modes**: `fetch` (Deno's native HTTP client) and `ffi-step` (Rust reqwest via the Step API)
- **Template engine**: `{{variable}}` substitution in URLs, headers, and bodies
- **Extractor system**: Capture values from JSON paths, response headers, regex groups, or DOM CSS selectors (via deno-dom) — stored in per-VU state for use in subsequent steps
- **Check system**: Eight check types including `status_in`, `body_includes`, `header_exists`, `header_equals`, `header_includes`, `json_path_exists`, `json_path_equals`, and `regex_match`
- **Cookie jar**: Full RFC-aware implementation in TypeScript (for `fetch` mode) with domain/path matching, expiry handling, and Secure attribute support

The `ffi-step` mode delegates cookie management and redirects to reqwest on the Rust side, while the `fetch` mode handles everything in TypeScript. Both modes share the same check and extractor evaluation logic.

The full scripting documentation is in loadgen-script

Example coverage

The `examples/` directory in loadgen-rs repository demonstrates the full feature surface:

Example	What it demonstrates
`simple.ts`	Minimal FFI benchmark (10 requests, single client)
`correlation_mvp.ts`	Login, extract JWT token, use in follow-up request
`cookie_redirect_local_mvp.ts`	Embedded Deno server with Set-Cookie + 302 redirect chain
`protocol-matrix.ts`	H1/H2/H3 batch + script mode for all three protocols
`ab-stages.ts`	Ramping VUs with stage interpolation (k6-like stages)
`native_step_skeleton.ts`	Direct Step API usage without the scenario runner
`all-functionality.ts`	Full integration test suite using Chai assertions

The `cookie_redirect_local_mvp.ts` example is particularly interesting: it spins up a local Deno HTTP server that sets cookies and issues redirects, then runs the script scenario against it — a self-contained integration test that validates the entire cookie + redirect + check pipeline.

Appendix: Commands and Reproducibility

Equivalent Command Pairs (h2load vs loadgen-rs)

To compare both tools fairly, use equivalent flags and remember: `--rps` is **per client** in both tools.

```
# 1) H1 count mode (same request budget)
h2load --h1 -n 200 -c 10 -t 8 -m 1 "https://bench.local:8082/?s=256k"
loadgen-rs --h1 -n 200 -c 10 -t 8 -m 1 "https://bench.local:8082/?s=256k"

# 2) H1 duration mode with pacing (~200 aggregate rps: 10 clients * 20
rps/client)
h2load --h1 -D 10s -c 10 -t 8 -m 1 --rps 20 "https://bench.local:8082/?s=
256k"
loadgen-rs --h1 --duration 10s -n 0 -c 10 -t 8 -m 1 --rps 20 "https://
bench.local:8082/?s=256k"

# 3) H2 duration mode with pacing (same effective target)
h2load --alpn-list=h2 -D 10s -c 10 -t 8 -m 1 --rps 20 "https://bench.
local:8082/?s=256k"
loadgen-rs --alpn-list h2 --duration 10s -n 0 -c 10 -t 8 -m 1 --rps 20
"https://bench.local:8082/?s=256k"

# 4) H3 duration mode with pacing (same effective target)
h2load --alpn-list=h3 -D 10s -c 10 -t 8 -m 1 --rps 20 "https://bench.
local:8082/?s=256k"
```

```
loadgen-rs --alpn-list h3 --duration 10s -n 0 -c 10 -t 8 -m 1 --rps 20
"https://bench.local:8082/?s=256k"
```

CLI Compatibility Map

The table below maps `loadgen-rs` options to their closest `h2load` equivalent.

loadgen-rs	h2load	Mapping	Notes
<URL>	<URI>...	partial	h2load accepts multiple URIs
-n <REQUESTS>	-n, --requests=<N>	same	
-D, --duration	-D, --duration	same	
--warm-up-time	--warm-up-time	same	
--ramp-up-time	none	no direct equivalent	Closest knobs are -r/--rate + --rate-period, but semantics differ
-c <CLIENTS>	-c, --clients=<N>	same	
-t <THREADS>	-t, --threads=<N>	same	
-m <MAX_STREAMS>	-m, --max-concurrent-streams=<N>	same	
--h1	--h1	same	
--alpn-list h2|h3	--alpn-list=<LIST>	partial	h2load allows arbitrary ALPN lists
--connect-timeout	none	no direct equivalent	Only related connection-level timeout controls exist (-T, -N)
--request-timeout	none	no direct equivalent	No per-request timeout flag in h2load
--tcp-quickack	none	none	
--method <METHOD>	none	no direct equivalent	h2load is primarily GET; POST is enabled via -d
-H, --header	-H, --header	same	
-d, --data <DATA>	-d, --data=<PATH>	partial	loadgen-rs: inline body; h2load: file path
--data-file <PATH>	-d, --data=<PATH>	close equivalent	
--rps <RPS>	--rps=<N>	same	per-client target in both tools (aggregate RPS * clients)
-k, --insecure	none	no CLI equivalent	No direct insecure toggle in h2load CLI
--tls-ciphers	--ciphers, --tls13-ciphers	partial	h2load splits TLS <=1.2 and TLS 1.3 ciphers
--tls-ca <PATH>	none	none	
-o, --output <PATH>	none	none	h2load offers --log-file (per-request TSV), not run-level JSON/CSV
--format jsonl|csv	none	none	
--no-tui	none	none	

Container Baseline

The multi-stage Containerfile builds the binary in a Rust image and copies it into a minimal Debian Slim image with `bash` and `curl`, with HTTP/3 support built in:

```
podman build -t loadgen-rs -f Containerfile .
podman run --rm --network host \
  --cap-add=SYS_ADMIN \
  --security-opt seccomp=unconfined \
  loadgen-rs -D 10s -c 8 -t 4 --alpn-list=h2 https://target:8443/
```

`--network host` avoids NAT overhead. `--cap-add=SYS_ADMIN` enables io_uring inside the container.

Test Server Setup

The test server used in this post is `haterm` from `HAProxy`, created and started with the following commands:

```
# create combined.pem in TLS-PATH
cat server.crt ca.crt server.key > combined.pem

# build container image
buildah bud -f Containerfile-hap-git -t hap-own

# run haterm
podman run -d --rm --name hap-own \
  --network host \
  --entrypoint=/usr/local/sbin/haterm \
  -v <TLS-PATH>/tls:/mnt:ro \
  localhost/haterm:latest \
  -L 127.0.0.1:8081 \
  -F 'bind quic4@127.0.0.1:8082 ssl crt /mnt/combined.pem alpn h3' \
  -F 'bind 127.0.0.1:8082 ssl crt /mnt/combined.pem alpn h2'
```

The used Containerfile-hap-git

Cloud Deployment (Terraform + Ansible)

Running distributed benchmarks on your local machines is useful for development, but real load testing demands dedicated cloud infrastructure — consistent hardware, dedicated NICs, and geographic proximity to the target. loadgen-rs ships with a complete Terraform + Ansible pipeline for Hetzner Cloud that takes you from zero to running workers in under five minutes.

The Terraform configuration provisions dedicated-vCPU instances (CCX series), places them in a private network, and configures a firewall that allows SSH and the worker-agent port. A single `generate-inventory.sh` script bridges the gap between Terraform and Ansible — it reads the Terraform outputs and writes a ready-to-use `ansible/inventory.ini` with the public IPs:

```
terraform apply
```

```
├── hcloud_server.worker[0]  →  49.13.x.x  (loadgen-worker-0)
├── hcloud_server.worker[1]  →  49.13.y.y  (loadgen-worker-1)
├── hcloud_network           →  10.0.1.0/24 (private)
└── hcloud_firewall          →  SSH + port 9091
│
▼
./generate-inventory.sh
│
▼
ansible/inventory.ini
├── loadgen-worker-0  ansible_host=49.13.x.x  worker_port=9091
└── loadgen-worker-1  ansible_host=49.13.y.y  worker_port=9091
│
▼
ansible-playbook deploy-workers.yml
│
▼
Workers ready — POST /run from controller
```

The full workflow from provisioning to results:

```
# 1. Provision Hetzner Cloud machines
cd terraform
cp terraform.tfvars.example terraform.tfvars
# Set: hcloud_token, ssh_key_name, worker_count=4, worker_type=ccx33
terraform init && terraform apply

# 2. Generate Ansible inventory from Terraform state
./generate-inventory.sh
# → ansible/inventory.ini with 4 workers

# 3. Deploy + benchmark + cleanup in one command
cd ..
scripts/distributed-remote.sh \
  -i ansible/inventory.ini \
  --target-url "https://bench.target:8443/?s=256k" \
  -c 1000 -D 60 --stop-after

# 4. Tear down when done
cd terraform && terraform destroy
```

The Ansible playbook (`deploy-workers.yml`) handles everything on the remote machines: installing Deno if not present, copying the FFI shared library and TypeScript files, deploying a systemd service unit (with a tmux fallback for systems without systemd), and running health checks to confirm each worker is ready. The matching `stop-workers.yml` playbook cleanly shuts everything down.

This separation of concerns — Terraform owns infrastructure, Ansible owns configuration, the shell script orchestrates the workflow — means you can mix and match. Already have machines? Skip Terraform, write `inventory.ini` by hand. Prefer a different cloud? Replace the `.tf` files, keep the same Ansible playbooks and scripts. The worker-agent doesn't care how it got deployed.

Component	Responsibility
`terraform/*.tf`	Provision Hetzner VMs, network, firewall
`terraform/generate-inventory.sh`	Bridge: Terraform outputs → Ansible inventory
`ansible/deploy-workers.yml`	Install Deno, deploy files, start workers
`ansible/stop-workers.yml`	Stop workers (systemd or tmux)
`scripts/distributed-remote.sh`	End-to-end: deploy → benchmark → stop

Conclusion

loadgen-rs is a lean, focused tool: it performs exactly one run, outputs the results as JSONL, and leaves the orchestration (warmup, parameter sweeps, plotting) to the calling script. The combination of raw h2, quinn, and rustls covers all three HTTP protocols — and thanks to per-worker runtimes and a lock-free design, the client itself is not the bottleneck.

CPU profiling confirmed this: over 60% of execution time is spent in TLS decryption, memory copies, and kernel syscalls — code paths shared by any HTTP benchmark client regardless of language. The remaining gaps to h2load are architectural properties of the Rust crate ecosystem (h2's async channel design, quinn's userspace QUIC overhead) rather than optimization opportunities in loadgen-rs itself.

With the Deno FFI layer, distributed mode, and Terraform/Ansible pipeline, loadgen-rs now serves five usage patterns: shell scripts that need a fast, h2load-compatible CLI tool; TypeScript users who want k6-style scripted scenarios with native H1/H2/H3 performance; multi-machine deployments where a single process can't generate enough load; cloud-provisioned benchmarks where you spin up dedicated Hetzner instances, run the test, and tear everything down in minutes; and distributed scripted scenarios where workers execute multi-step request sequences with checks and response validation on remote machines. The two-API design (batch for throughput, step for correlation) plus the controller/worker/infrastructure stack bridges all five without compromising any of them.

Code and Repository Link

The code is on loadgen-rs - GitHub — PRs welcome.