envoycon

From Microbenchmarks to HTTP2 Load-testing: 5 Performance Tools and Techniques to Improve Envoy Scalability

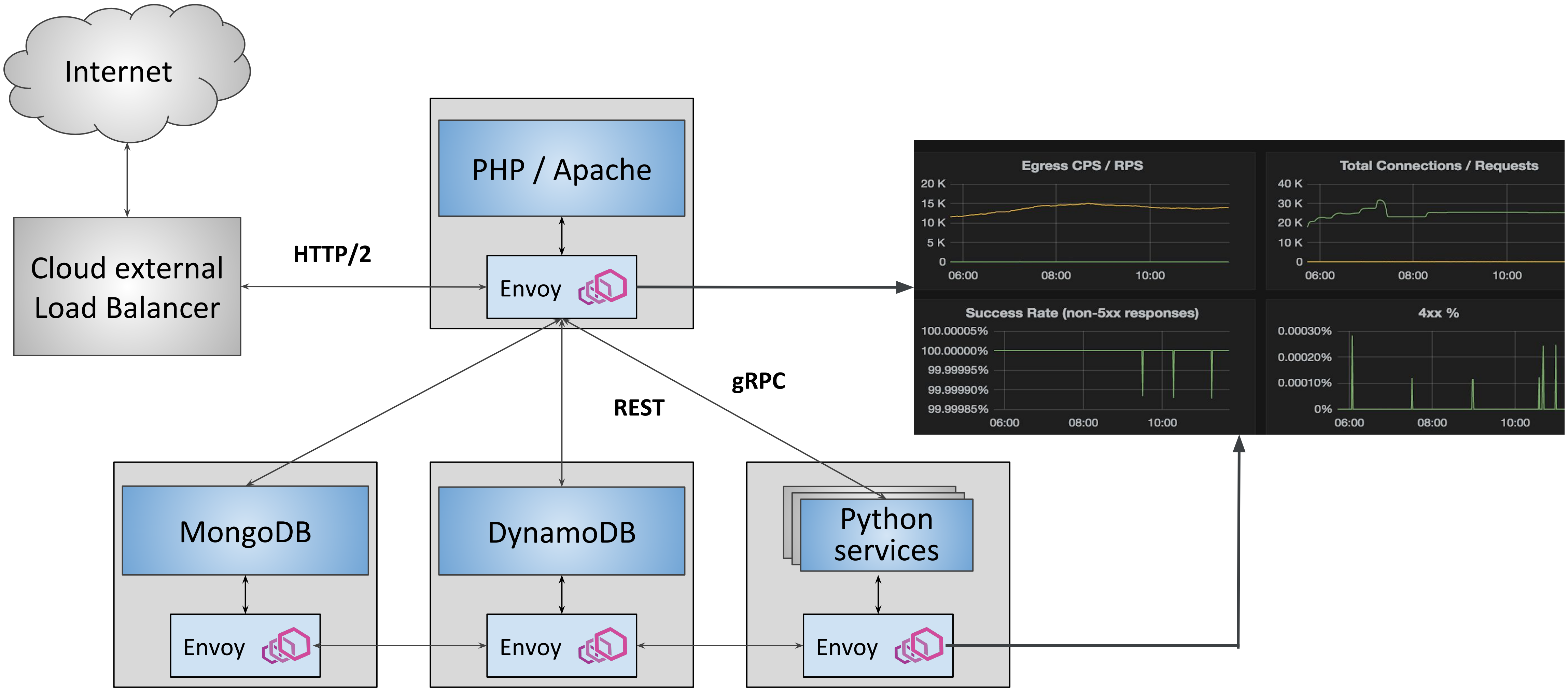
Joshua Marantz (Google, @jmarantz)

Otto van der Schaaf (We-Amp. @oschaaf)

Talk Structure

- **A Very Brief Intro to Envoy**
- Situational use of tools and techniques
  - Haphazard
  - Performance Annotations
  - Microbenchmarks
  - Application-level load testing
  - Fuzzing for Performance

Adding Observability to a Service Oriented Architecture



Talk Structure

- **Best-in-class observability**
- **Performance**
- Reliability
- Modern codebase
- Extensibility
- Configuration API
- **Community**

So… how does Envoy scale?

Scaling Dimensions

- Initialization performance
- Memory footprint
- Sensitivity to configuration size
- Sensitivity to traffic
- Sensitivity to large numbers of cores
- Data plane performance: Latency, Throughput

What about Envoy in Multi-tenant scenarios?

- Envoy as a front-end Proxy
- Cloud Providers
- CDNs

All these scenarios may define 10+k “clusters” (backend service groups), depending on scale

Situational use of tools and techniques

- **Haphazard**
- Performance Annotations
- Microbenchmarks
- Application-level load testing
- Fuzzing for Performance

Case-study: stat scalability issues during startup

- $O((100*\mathit{numClusters})^2)$ startup issue (#2063)
  - Very easy to find using **haphazard** methods
    - ^C in the debugger
    - Or any other method of strobing active threads
- 20x speed up @10k clusters via better data structures *(#2358)*
- >10 minutes → 22 seconds; still too slow

  → *mission not yet accomplished*

Stat scalability phase 2: regex tag init bottleneck

- Also easy-to-find haphazardly or via profiling: #2373
- Possible causes may include:
  - Regex package is slow (std::regex is much slower than RE2)
  - Too many regex lookups (lookup count found with **callgrind**)
  - Very complicated regexes, some with **catastrophic backtracking**
- Regexes were used to add tagging structure to scalar stats
  - Needed for several stats sinks
- Solution was less obvious

Performance Annotations

Performance macro library, enabled by compile flag: [envoy proxy perf_annotation.h]

Initiates a performance operation, storing its state in perf_var.

```
PERF_OPERATION(perf_var)
```

Records performance data initiated with PERF_OPERATION.

```
PERF_RECORD(perf_var, category, description)
```

Dumps recorded performance data to stdout. Expands to nothing if not enabled.

```
PERF_DUMP()
```

Performance Annotations in stats tag extraction

```
for (extractor : all_tag_matcher_regexes_) {
    PERF_OPERATION(perf);
    std::smatch match;
    if (std::regex_search(stat_name, match, extractor.regex) &&
        match.size() > 1) {
        /* Update tags, extract tags from stat_name */
        PERF_RECORD(perf, "re-match", extractor.pattern);
    } else {
        PERF_RECORD(perf, "re-miss", extractor.pattern);
    }
}
PERF_DUMP()
```

Annotated path; static in code

Data-derived annotation

Related work: VMware kstats, https://dl.acm.org/citation.cfm?id=1899945

10k clusters regex vie itialization

```
Duration(us)  # Calls  Mean(ns)  StdDev(ns)  Min(ns)  Max(ns)  Category  Description
     1252735   910190      1376     483.682      242    47638  re-miss   envoy.grpc_bridge_method
     1233659   910190      1355     453.037      234    28921  re-miss   cipher_suite
     1216013   910190      1335         452      232    54768  re-miss   envoy.grpc_bridge_service
      941602  1820243       517     279.096      238    27092  re-miss   envoy.http_conn_manager_prefix
      749437   910190       823     361.951      261    98978  re-miss   envoy.response_code
      695964   910170       764       393.6      238    43740  re-miss   envoy.respo
...................................elided 12 similar rows, each costing > 0.6 seconds real time.............
      624921   910190       686     318.941      234    28680  re-miss   envoy.virtu
      485891   910190       533      278.86      239    41152  re-miss   envoy.worke
      474777   910190       521     272.132      227    36750  re-miss   envoy.mongo
      474536   910190       521     264.689      233    23060  re-miss   envoy.rds_ro   onfig
      473125   910184       519     263.062      236    24118  re-miss   envoy.li   tener_address
      464885   910190       510     270.477      228    28184  re-miss   envoy.virtual_host
      426139   910000       468     276.855      407    26796  re-match  envoy.cluster_name
         110                           .535      226     1086  re-miss   envoy.cluster_name
          65        1                  .711      283     1791  re-match  envoy.http_conn_manager_prefix
          15        2                  .737      490     1066  re-match  envoy.response_code_class
           3                           .418      401     1411  re-match  envoy.listener_address
```

3 regexes were much more expensive per/eval than the others

Almost all high-cost regex operations were mismatches

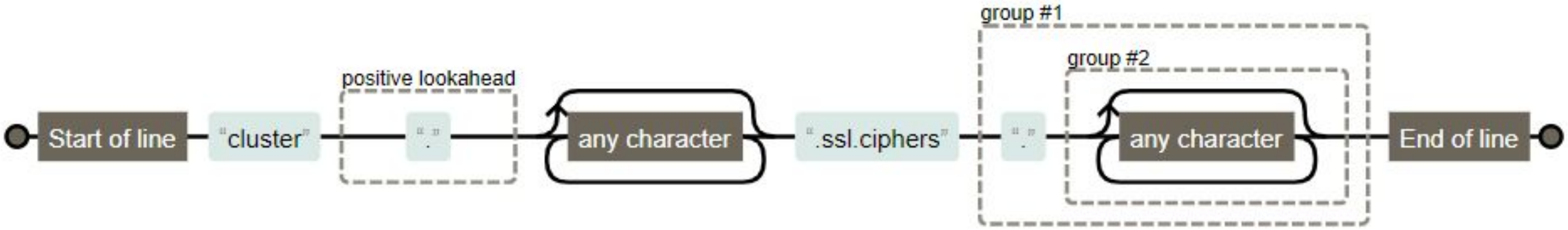
Only one regex match took material amounts of time: 0.4 sec

Total mismatch time: 17 seconds, or 76% of init time

Prefiltering to reduce the number of regex evals

Sample tag-extraction regex:

```
^cluster(?=\.).*?\.ssl\.ciphers(\.(.*?))$
```



- Must begin with "**cluster.**" → build a map<prefix, RegexList>
- Must also contain "**ssl.ciphers**" → screen for substrings prior to regex

Performance Annotations with prefix filtering

```
init() { computePrefixToRegexListMatch(); }

for (extractor : findPossibleExtractorsWithPrefix(stat_name)) {
  PERF_OPERATION(perf);
  if (extractor.substrMismatch(stat_name)) {
    PERF_RECORD(perf, "re-skip", name_);
    continue;
  }
  std::smatch match;
  if (std::regex_search(stat_name, match, extractor.regex) &&
      match.size() > 1) {
    /* Update tags, tag_extracted name */
    PERF_RECORD(perf, "re-match", extractor.pattern);
  } else {
    PERF_RECORD(perf, "re-miss", extractor.pattern);
  ...
```

10k clusters with prefiltering: 3.75s initialization

```
Duration(us)  # Calls  Mean(ns)  StdDev(ns)  Min(ns)  Max(ns)  Category  Description
      372381   910000       409     282.391      370    28669  re-match  envoy.cluster_name
      122926   160054       768     335.647      487    44714  re-miss   envoy.response_code
      121003   160034       756     422.152      482    28839  re-miss   envoy.response_code_class
       41140   750136        54     82.8142       21    26367  re-skip   envo
       39580   750136        52     79.6177       21    15930  re-skip   envo
       36000   910000        39     77.9917       28    31885  re-skip   ciph_
       34498   910000        37     77.3431       27    25672  re-skip   envoy.grpc_bridge_service
       32838   910000                             27    16060  re-skip   envoy.grpc_bridge_method
          66      137                             33     5667  re-match  envoy.http_conn_manager_prefix
          31       21                             82     3788  re-miss   envoy.ssl_cipher
          15       20       768     225.804      472     1470  re-match  envoy.response_code_class
          11       15       747     169.845      614     1304  re-miss   envoy.listener_address
           5      125        46      23.967       30      249  re-skip   envoy.dynamo_partition_id
           5      125        44     16.2906       30      156  re-skip   envoy.dynamo_operation
           5      125        42     11.1086       29      124  re-skip   envoy.rds_route_config
           5      125        40     8.34395       28       65  re-skip   envoy.fault_downstream_cluster
           4      125        39     10.2826       29      107  re-skip   envoy.dynamo_table
           4      125        38      8.6811       30      112  re-skip   envoy.http_user_agent
           3        6       643     418.244      415     1487  re-match  envoy.listener_address
           0        9        70     11.9269       47       84  re-skip   envoy.http_conn_manager_prefix
           0        9        33     15.1033       28       74  re-skip   envoy.worker_id
```

Most time-consuming regex is now a MATCH

Most regexes consume < 0.05 seconds

Next steps toward improving performance

- Improving per-regex cost
  - Simplifying regular expressions so they evaluate faster
  - Switching to a better library, such as Google's RE2
    - `std::regex` allocates memory during pattern matching
    - Can run out of stack space on large input
- Enter Microbenchmarks ...

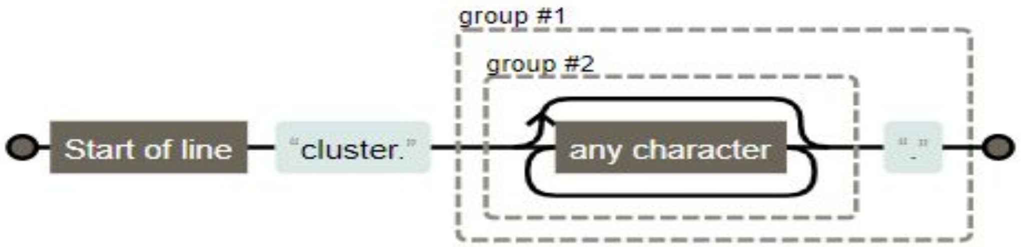
Microbenchmarks

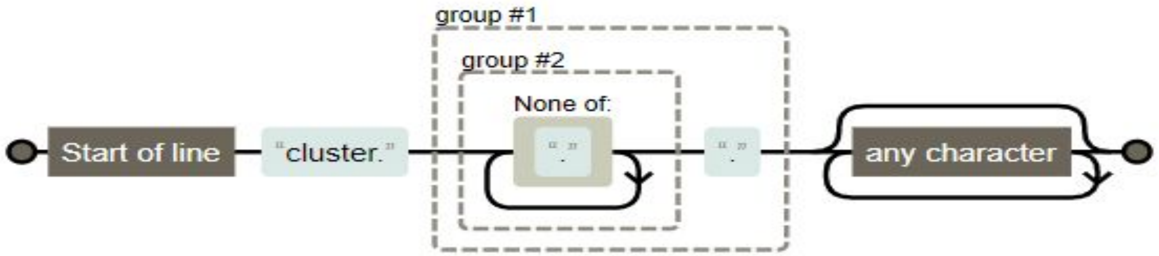
- A benchmark designed to measure the performance of a very small and specific piece of code.
  - [google microbenchmark]
  - [alternative microbenchmark libraries]

- Useful at multiple phases development
  - Weighing algorithm choices when writing code
  - Improving performance of a block of code known to be hurting performance

Microbenchmark for regexes

```
static void BM_StdRegex(benchmark::State& state)
 std::regex re(“^cluster\\.((.*?)\\.)”);
  // alternate: ^cluster\\.(([^\\.]+)\\.).*
  for (auto _ : state) {
   for (const std::string& cluster_input : inputs) {
     std::smatch match;
     if (std::regex_search(cluster_input, match, re)) {
       ++passes;
     }
   }
 }
}
BENCHMARK(BM_StdRegex);
```





https://github.com/jmarantz/envoy/blob/re-speed-benchmark/test/common/common/re_speed_test.cc

Microbenchmarks comparing variants

```
--------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations
--------------------------------------------------------------------------
BM_StdRegex                           1339 ns         1339 ns       441023
BM_StdRegexStringView                 1288 ns         1288 ns       544246
BM_StdRegexStringViewAltPattern       1980 ns         1980 ns       354793
BM_RE2                                1050 ns         1050 ns       666026
BM_RE2_AltPattern                      448 ns          448 ns      1564090
```

- Test-patterns informed by given prefix-filtering (mismatches rare)
- std::regex improved 5% by using string_view rather than std::string
- RE2 on same regex improved another 25%
- RE2 with a better pattern provided a further 55% reduction, though that same pattern made std::regex worse

10% speed-up (3.75s -> 3.44s) with 1 new regex

```
Duration(us)  # Calls  Mean(ns)  StdDev(ns)  Min(ns)  Max(ns)  Category   Description
      184379   910000       202     230.157      145   137020  re2-match  envoy.cluster_name
      124345   160054       776     343.242      499    27504  re-miss    envoy.response_code
      118753   160034       742      399.93      495    17136  re-miss    en                     s
       42086   750136        56     71.8637       24    16121  re-skip    en
       39155   750136        52     76.7944       25    16226  re-skip    en                     s
       35589   910000        39     150.468       28   131877  re-skip    cipher_suite
       33933   910000        37     65.1851       27    16145  re-skip    envoy.grpc_bridge_service
       32949   910000        36     82.8183       27    25299  re-skip    envoy.grpc_bridge_method
          61      137       451     399.466      244     3560  re-match   envoy.http_conn_manager_prefix
          30       21      1469     494.582     1007     3295  re-miss    envoy.ssl_cipher
          16       20       801     221.848      460     1427  re-match   envoy.response_code_class
          11       15       757     137.208      637     1199  re-miss    envoy.listener_address
           5      125        46     24.9203       30      242  re-skip    envoy.dynamo_partition_id
           5      125        43     16.6237       30      155  re-skip    envoy.dynamo_operation
           5      125        41     9.41567       29      101  re-skip    envoy.rds_route_config
           5      125        40     15.0171       28      154  re-skip    envoy.fault_downstream_cluster
           4      125        38     10.9989       28      112  re-skip    envoy.http_user_agent
           4      125        38     10.9942       29      114  re-skip    envoy.dynamo_table
           3        6       636     405.296      454     1463  re-match   envoy.listener_address
           0        9        66     17.7795       30       92  re-skip    envoy.http_conn_manager_prefix
```

Pattern accelerated from 0.37 seconds to 0.184 seconds

https://github.com/envoyproxy/envoy/pull/8831

Take-aways: Microbenchmarks and Annotation

- ***Run the system to learn where the bottlenecks are***
- Annotate if needed to understand data-dependent issues
- Microbenchmark that data to converge to a solution
    - Easier to see small changes at nanosecond level
    - Faster iteration

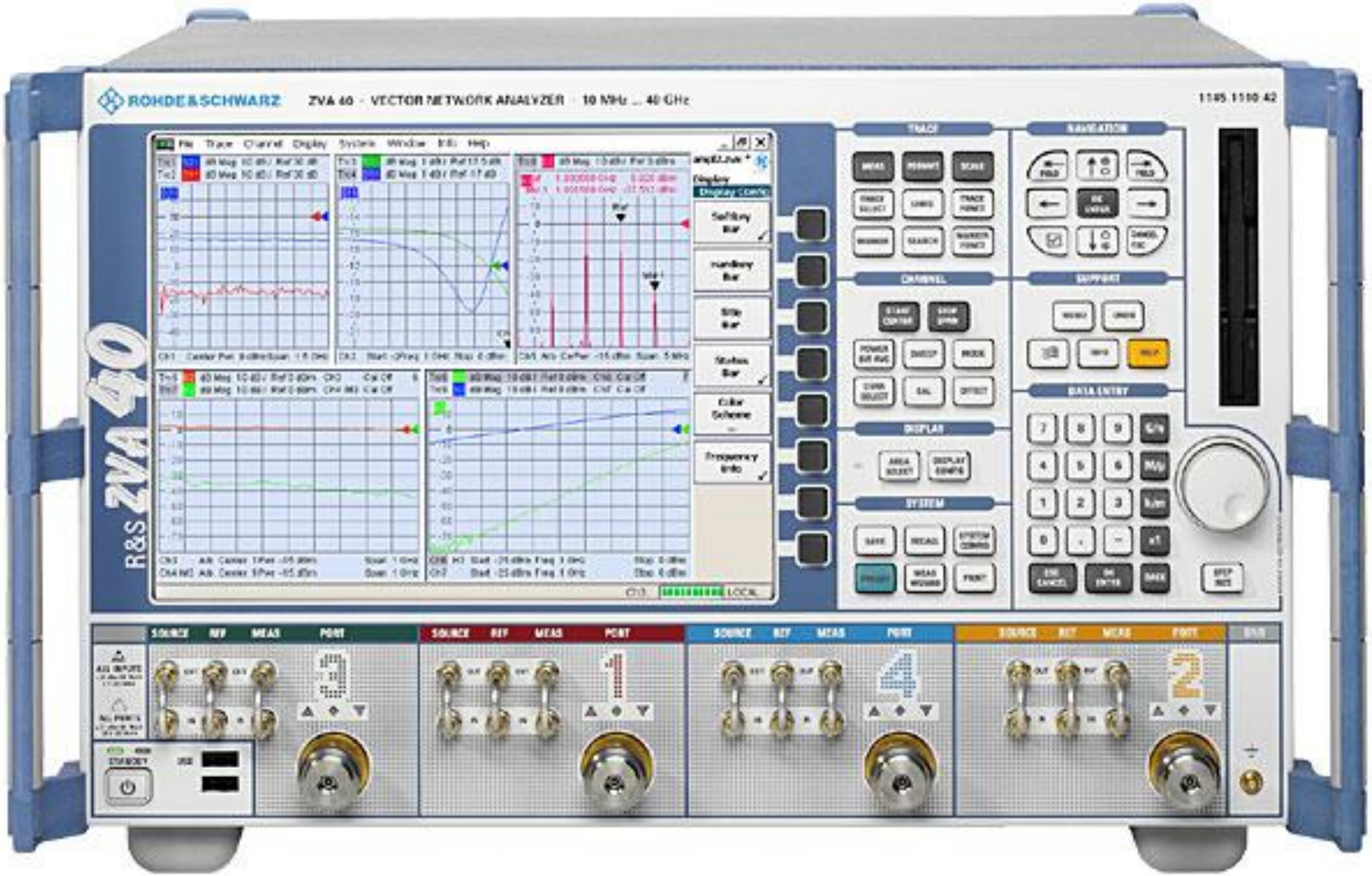
Situational use of tools and techniques

- Haphazard
- Performance Annotations
- Microbenchmarks
- **Application-level load testing**
- Fuzzing for Performance

Performance-testing for an L7 Proxy

Testing tools must be ≥ the performance of the system they are measuring:

- Capture long-tail latency
- 10-50 µs resolution
- < 5% variance
- Native http2
- API/command-line



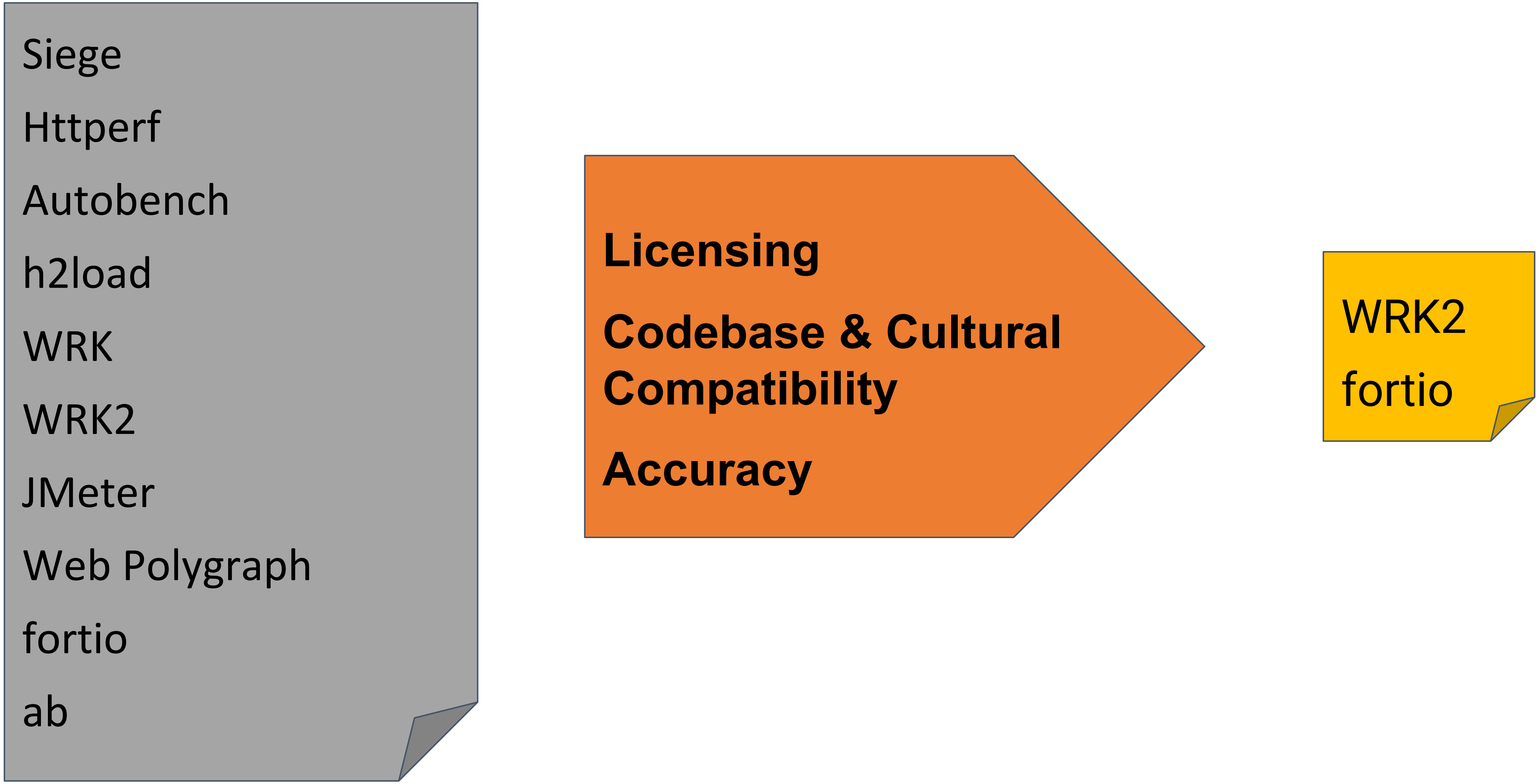
https://commons.wikimedia.org/wiki/File:R%26S_ZVA_40_4.jpg

What HTTP(2) load generators are there?

HTTP(2) load-generator requirements

- 10-50 µs resolution
- Envoy-compatible engineering standards:
  - Test coverage, code reviews, continuous integration...
- HTTP/1 and HTTP/2 support, path to HTTP/3
- Open loop and closed loop modes
- Load-generation server via gRPC
- Latency histograms
- Structured output formatting
- OSS license compatible with Envoy’s needs

Load Generation Evaluation funnel



Siege

Httperf

Autobench

h2load

WRK

WRK2

JMeter

Web Polygraph

fortio

ab

Licensing

Codebase & Cultural Compatibility

Accuracy

WRK2

fortio

Detailed Evaluation: WRK2

- ➕ Good at spreading load
- ➖ No HTTP2
- ➖ No test coverage
- ➖ Inactive community
- ➖ Sampling only
- ➖ Locking while sampling latency
- ➖ Millisecond granularity limitations

Detailed Evaluation: fortio

- ➕ Supports HTTP2
- ➕ Very nice visualizations
- ➖ Floating point math issues
- ➖ Go Garbage collection introduces jitter at µs granularity
- ➖ Governing scheduler (no fairness guarantees)
- ➖ Order of magnitude less accurate than WRK2 & Envoy Access logs control
- ➖ Uneven (batched) request-release timings

Decision: build a load-generator on top of Envoy

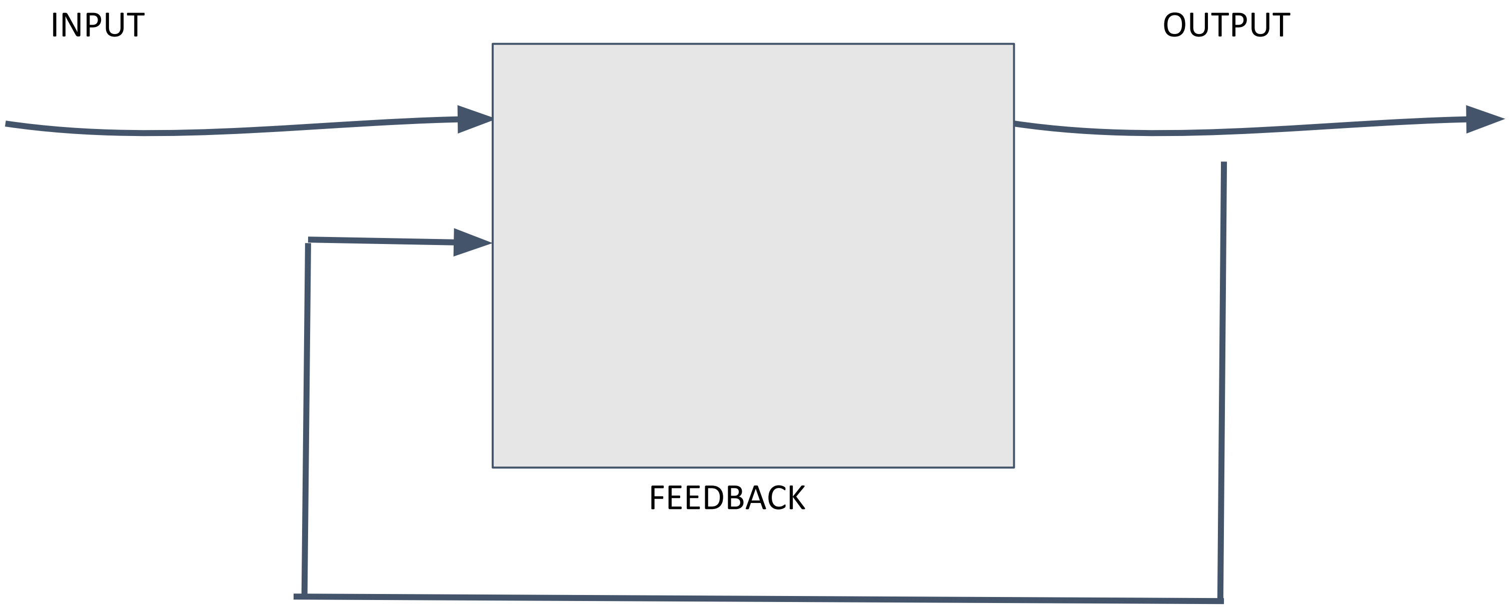
- ➕ Supports HTTP2
- ➕ Inherits HTTP3/Quic support when available in Envoy
- ➕ High accuracy & scalability
- ➕ Benefits from years of internal load-generation experience
- ➕ Can establish great testing / code review / CI culture
- ➕ Visualization agnostic
- ➖ Building takes time
- ➖ Building costs money
- ➖ Will require continued investment

Nighthawk: a load generator based on Envoy’s Network stack

Nighthawk: an OSS H2/http traffic generator

- Built on Envoy network stack
- Github repo (parallel to Envoy, dependent on it)
- Envoy style test coverage, CI, C++ style, code reviews
- Performance knobs
  - Http2 vs Http
  - Max Active Requests
  - Concurrency
  - Targeted requests-per-second
- Outputs latency histograms, other stats

Nighthawk: closed-loop mode



- Resource limits will induce back-pressure and influence request timings

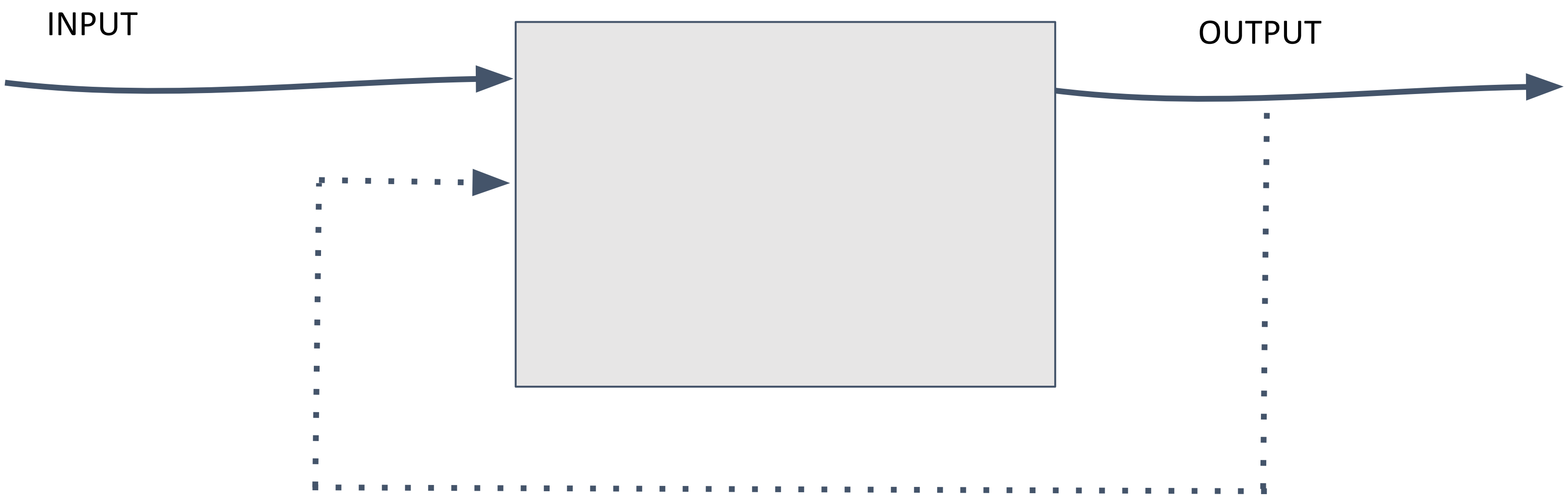
- Wait time for configured resource limits reported as “blocking”

Nighthawk: command line closed-loop test

```
taskset -c 0 ./nighthawk_client --concurrency auto --rps 15000 --duration 10 --connections 1 127.0.0.1:10000

[10:11:30.169193][5235][I] [source/client/process_impl.cc:170] Detected 1 (v)CPUs with affinity..

....

Request start to response end
  samples: 149942
  mean:    0s 000ms 037us
  pstdev:  0s 000ms 003us

  Percentile  Count       Latency
  0           1           0s 000ms 034us
  0.5         74976       0s 000ms 037us
  0.75        112471      0s 000ms 037us
  0.8         120045      0s 000ms 037us
  0.9         134980      0s 000ms 038us
  0.95        142447      0s 000ms 041us
  0.990625    148541      0s 000ms 049us
  0.999023    149796      0s 000ms 067us
  1           149942      0s 000ms 572us

Counter                                 Value       Per second
benchmark.http_2xx                      149943      14994.29
upstream_cx_http1_total                 1           0.10
upstream_cx_rx_bytes_total              24440709    2444069.14
upstream_cx_total                       1           0.10
upstream_cx_tx_bytes_total              8996580     899657.35
upstream_rq_total                       149943      14994.29
```

```
Blocking. Results are skewed when significant numbers are reported here.
  samples: 241
  mean:    0s 000ms 042us
  pstdev:  0s 000ms 006us

  Percentile  Count       Latency
  0           1           0s 000ms 037us
  0.5         121         0s 000ms 040us
  0.75        181         0s 000ms 041us
  0.8         193         0s 000ms 041us
  0.9         217         0s 000ms 045us
  0.95        229         0s 000ms 050us
  0.990625    239         0s 000ms 054us
  1           241         0s 000ms 128us
```

Nighthawk: open-loop mode



- No feedback-loop based on resource limits. Requests are released unconditionally when they are due.
- Counters will track failures because of configured Nighthawk resource limits.

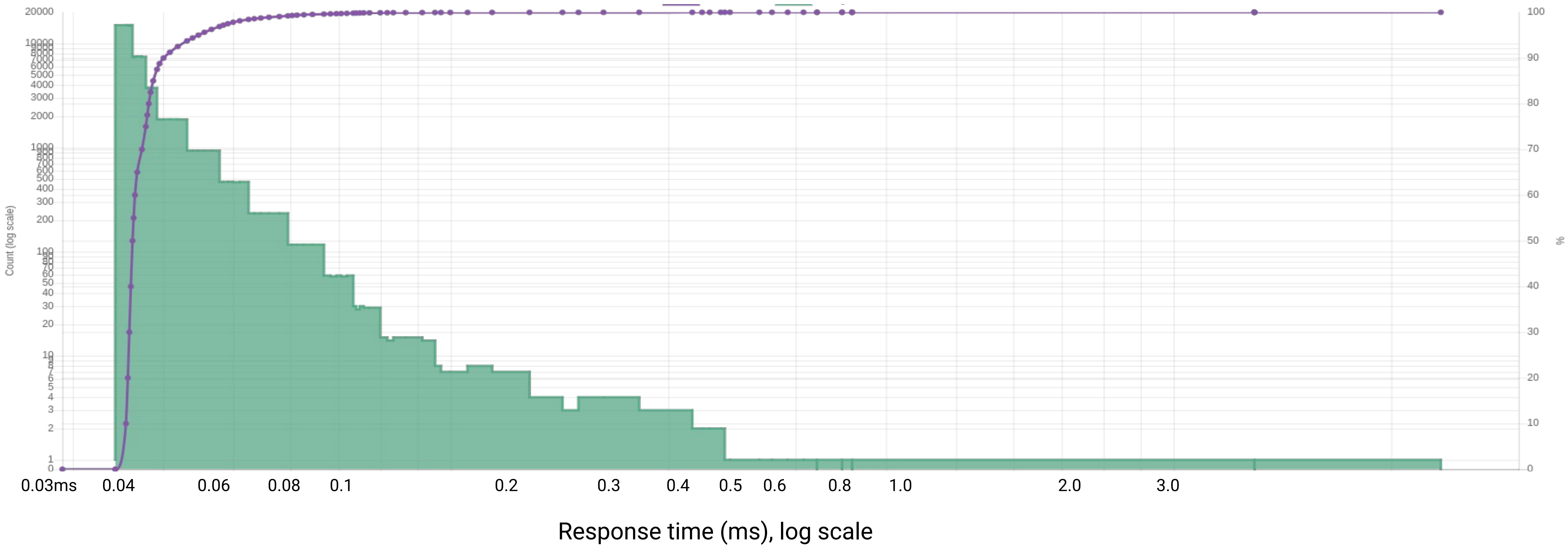
Nighthawk: command line open-loop test

```
taskset -c 0 ./nighthawk_client --concurrency auto --rps 15000 --duration 10 --connections 1 --open-loop 127.0.0.1:10000

[10:19:10.081859][6043][I] [source/client/process_impl.cc:170] Detected 1 (v)CPUs with affinity..

...

Request start to response end
  samples: 149849
  mean:    0s 000ms 038us
  pstdev:  0s 000ms 004us

  Percentile  Count       Latency
  0           1           0s 000ms 036us
  0.5         75085       0s 000ms 038us
  0.75        112526      0s 000ms 038us
  0.8         120009      0s 000ms 038us
  0.9         134915      0s 000ms 038us
  0.95        142360      0s 000ms 039us
  0.990625    148446      0s 000ms 043us
  0.999023    149703      0s 000ms 061us
  1           149849      0s 001ms 408us

Counter                                 Value       Per second
benchmark.http_2xx                      149850      14984.97
upstream_cx_http1_total                 1           0.10
upstream_cx_overflow                    6           0.60
upstream_cx_rx_bytes_total              24425550    2442549.56
upstream_cx_tx_bytes_total              8991000     899098.00
upstream_rq_total                       149850      14984.97
```

Nighthawk: zooming in

Export to **Fortio** for close inspection of the long tail. It's unexpectedly long.

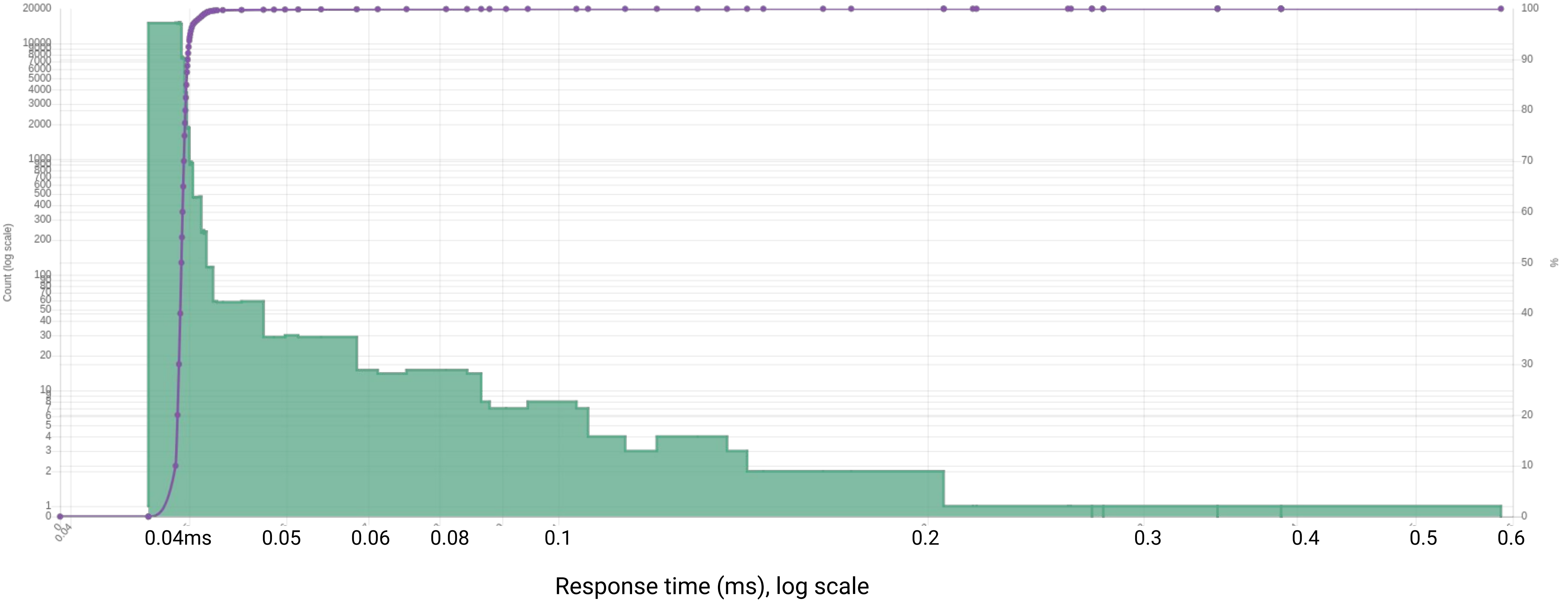
```
./nighthawk_client .. --output-format fortio > export-to-fortio.json
```



Nighthawk: single threaded pinned test server

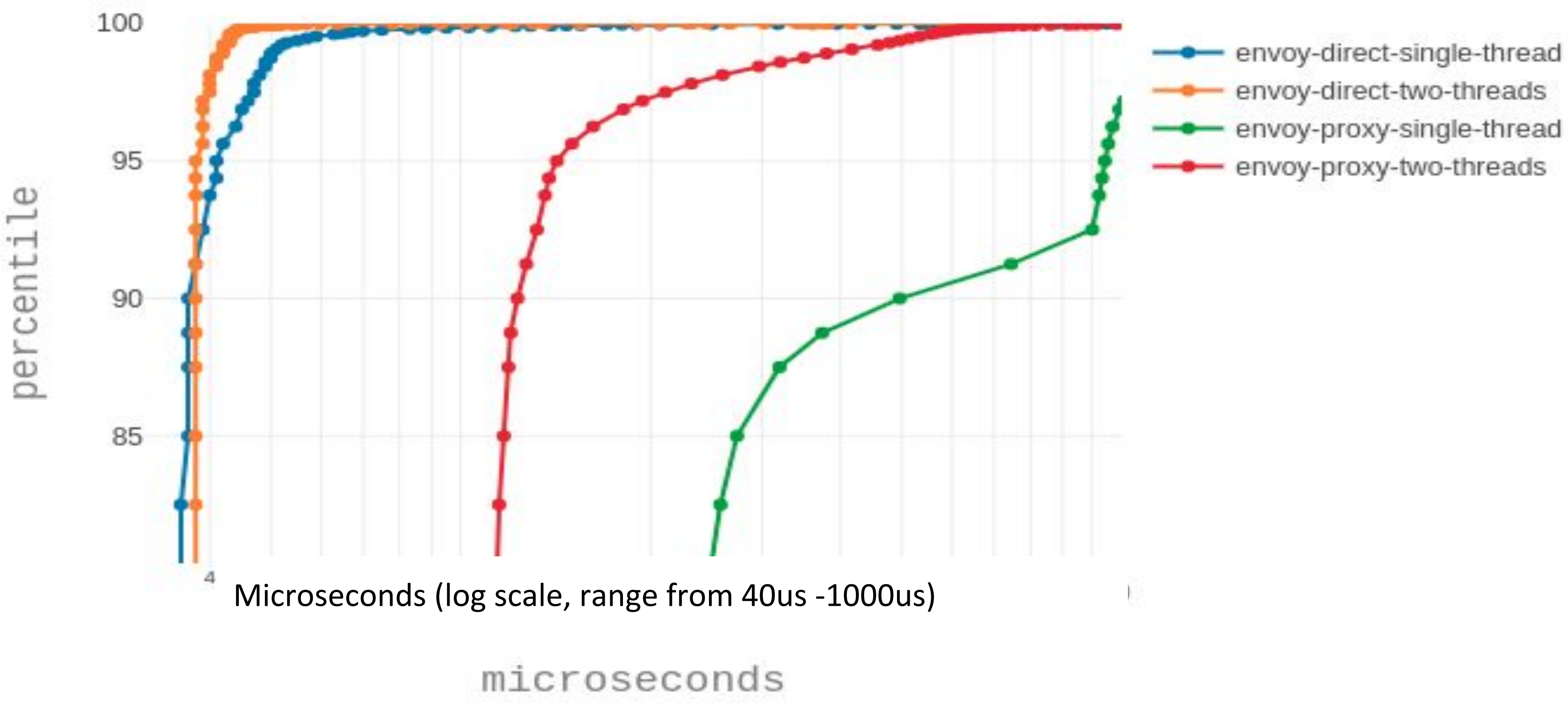
Pinning test-server to a CPU has a dramatic impact

```
taskset -c 39 bazel-bin/nighthawk_test_server  -c ~/envoy.yaml
```



Nighthawk: Measuring small differences

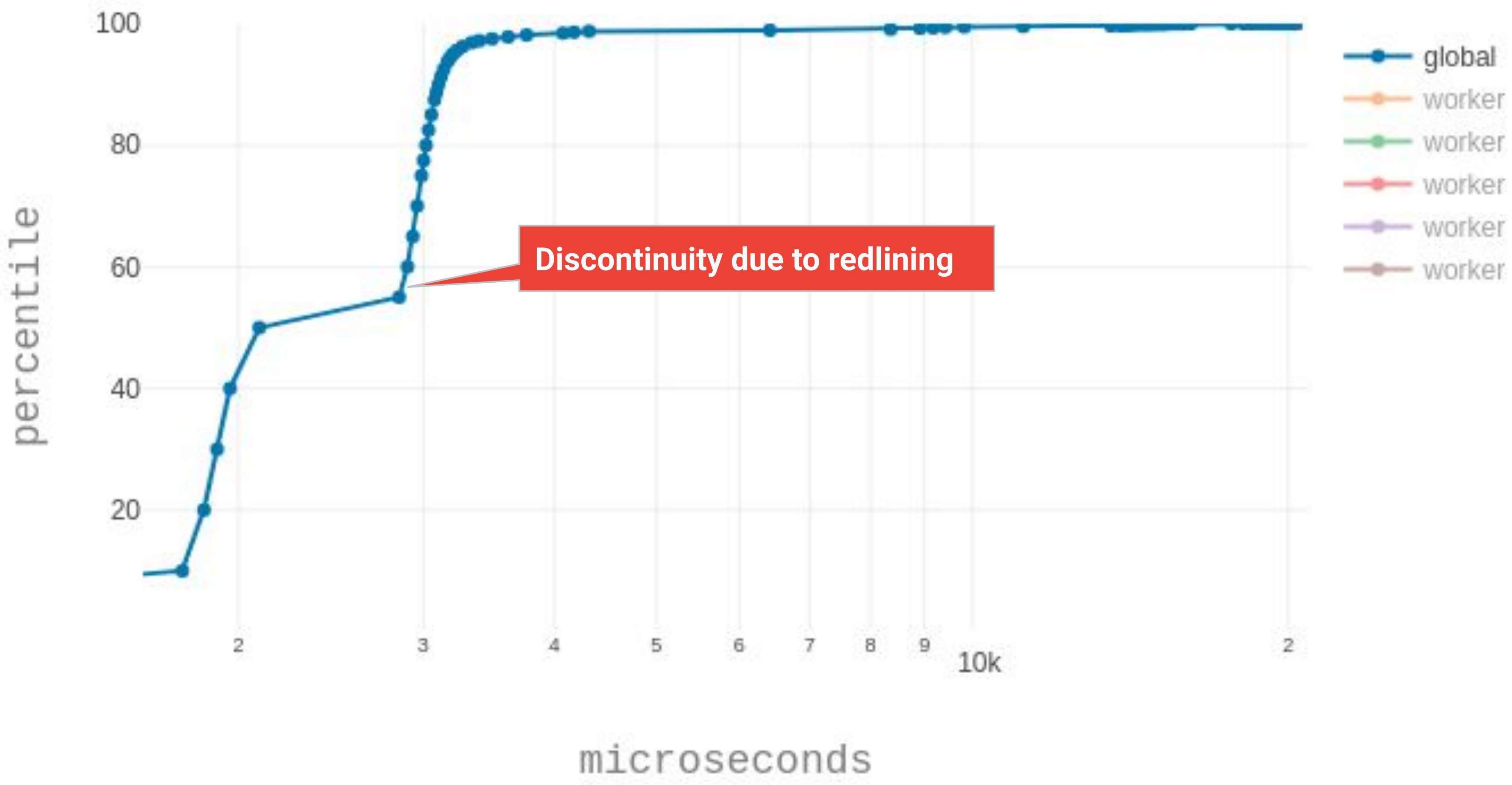
- Envoy in direct response mode vs. through Envoy in proxy mode
- Single-threaded Envoy vs double-threaded Envoy
- Small request / reply size, H1, perfect keep-alive @ 15K RPS



Microseconds (log scale, range from 40us -1000us)

Nighthawk: per-worker reporting

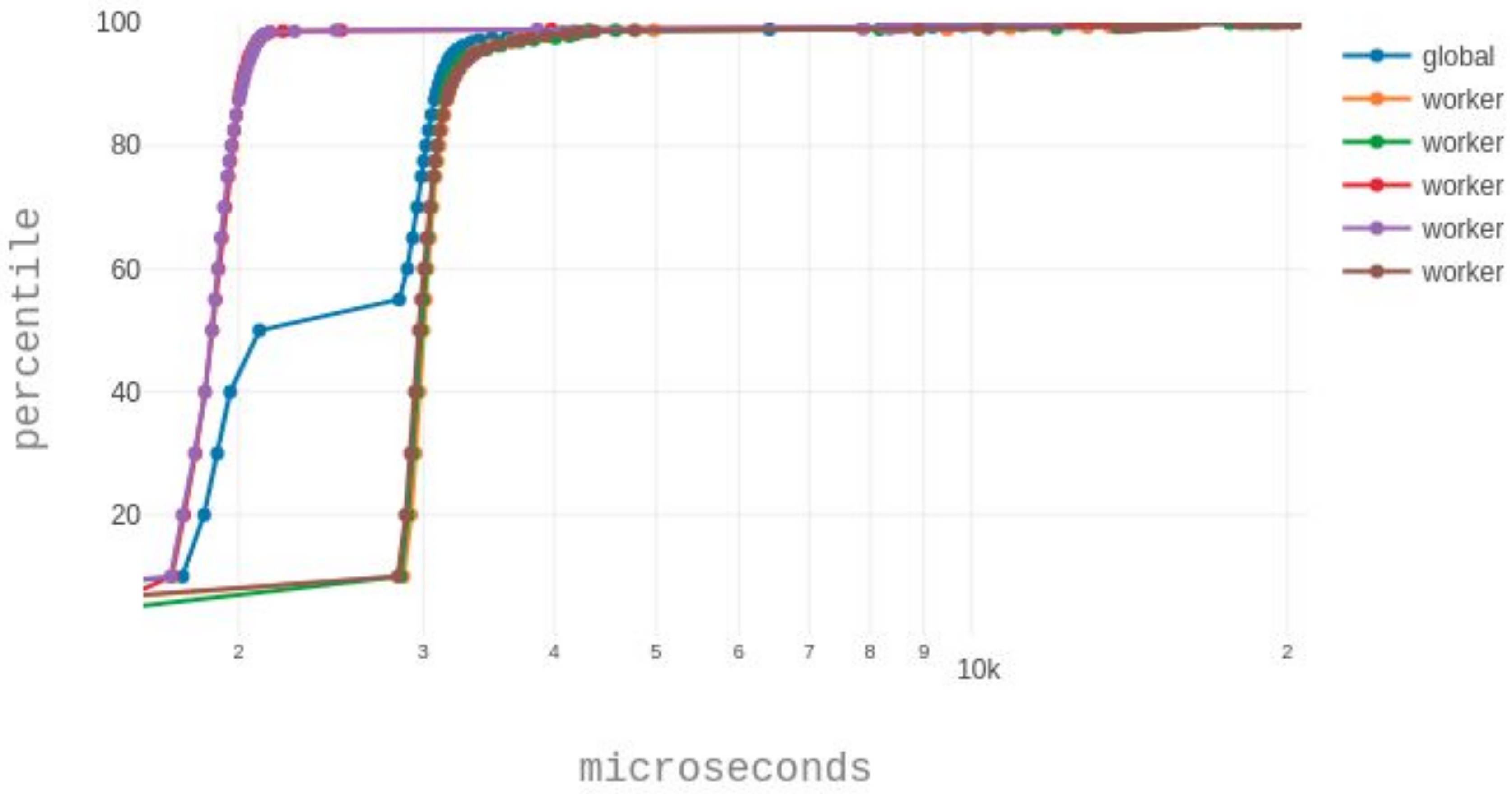
- Redline testing example: server on 2 cpu cores
- Nighthawk on 5 cores



Discontinuity due to redlining

Nighthawk: per-worker reporting

The per worker visualization show imbalances, explaining the odd shape of the aggregated result. Potential backend process hotspotting.



Nighthawk: Envoy H1 vs H2

Nighthawk’s facilitates writing targeted benchmarks with a small python framework .
This will capture profiling data as well as yield structured output in json.

Let’s compare downstream H1 vs H2(C) performance for small request / replies via closed-loop high rps tests.

Sample test scripts

```
def test_h1_concurrent_redline(http_test_server_fixture):
  http_test_server_fixture.test_server.enableCpuProfiler()
  parsed_json, _ = http_test_server_fixture.runNighthawkClient(
      [http_test_server_fixture.getTestServerRootUri(), "--rps", "50000", "--duration", "60", "--concurrency 120",
"--connections", "100", "--max-pending-requests", "100"])

def test_h2_concurrent_redline(http_test_server_fixture):
  http_test_server_fixture.test_server.enableCpuProfiler()
  parsed_json, _ = http_test_server_fixture.runNighthawkClient(
      [http_test_server_fixture.getTestServerRootUri(), "--rps", "50000", "--duration", "60", "--concurrency 120", "--h2",
"--max-pending-requests", "100"])
```

Command to execute:

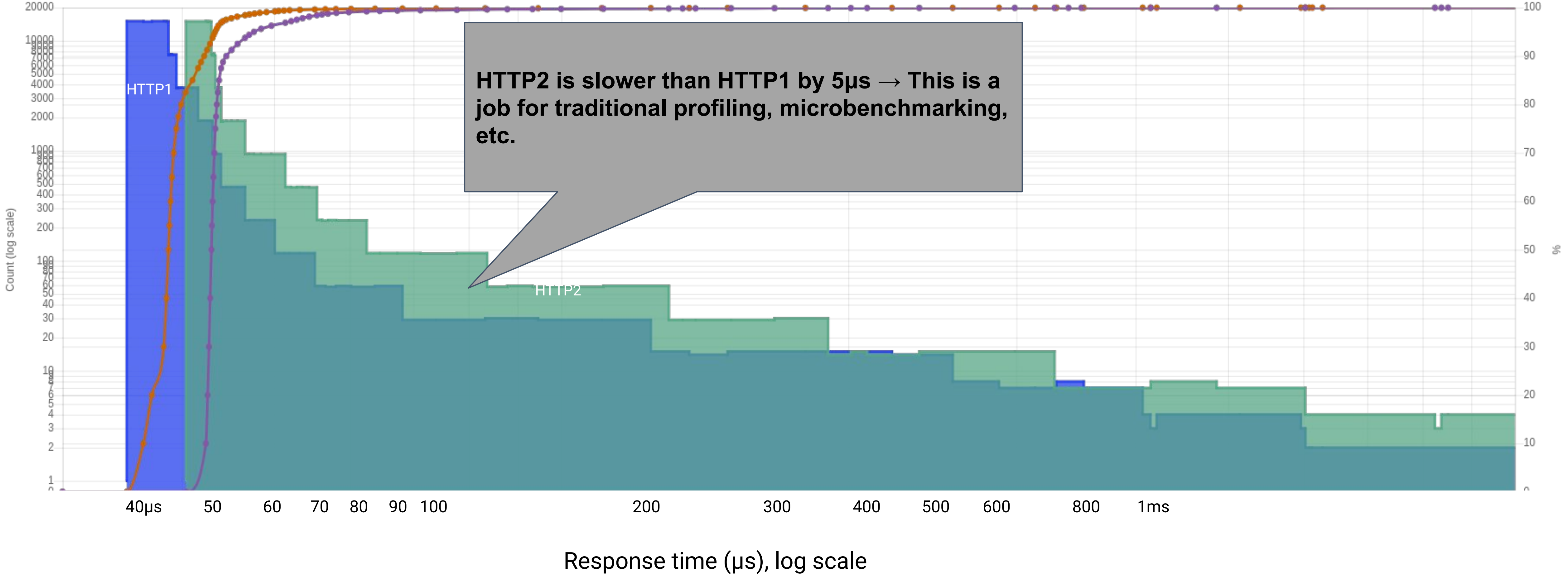
```
bazel test --test_env=ENVOY_IP_TEST_VERSIONS=v4only --test_env=HEAPPROFILE=
--test_env=HEAPCHECK= --cache_test_results=no --compilation_mode=opt --cxxopt=-g --cxxopt=-ggdb3
//benchmarks:*
```

HTTP1

```
nighthawk_client  --rps 15000
--connections 10 …
```

HTTP2

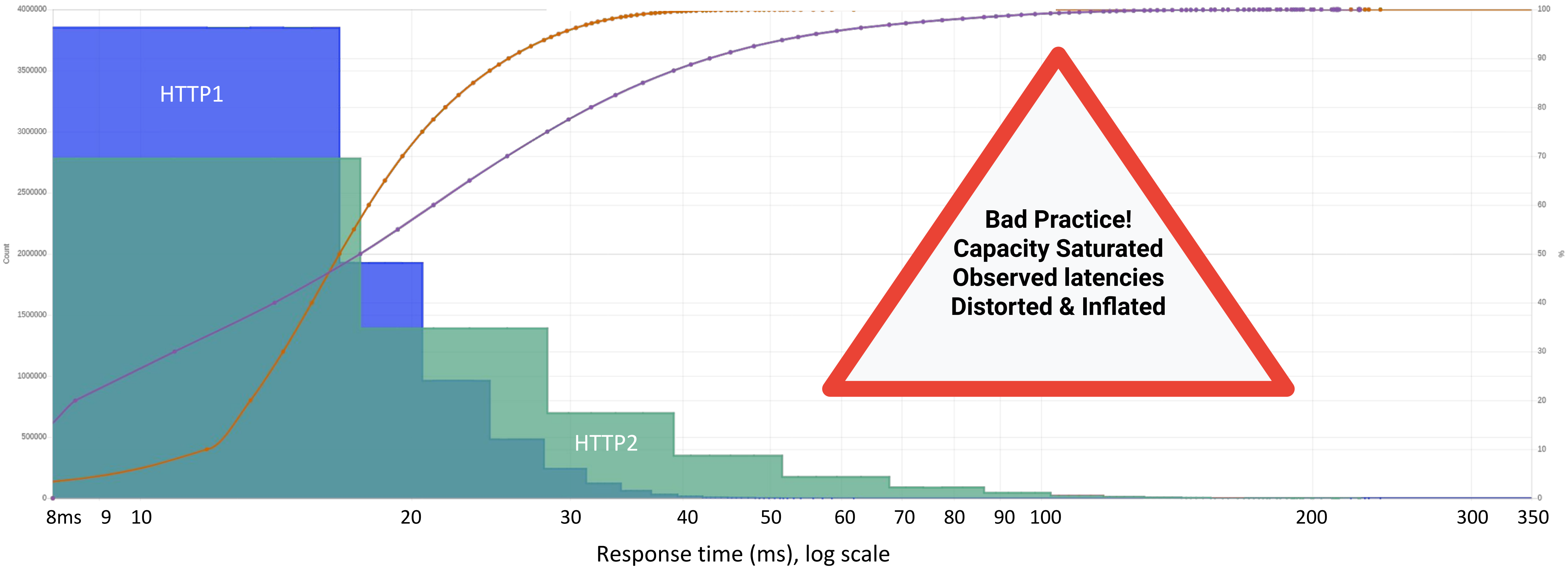
```
nighthawk_client -h2 --rps 15000
--max-active-requests 10 …
```



HTTP2 is slower than HTTP1 by 5µs → This is a job for traditional profiling, microbenchmarking, etc.

Nighthawk: Envoy H1 vs H2 performance

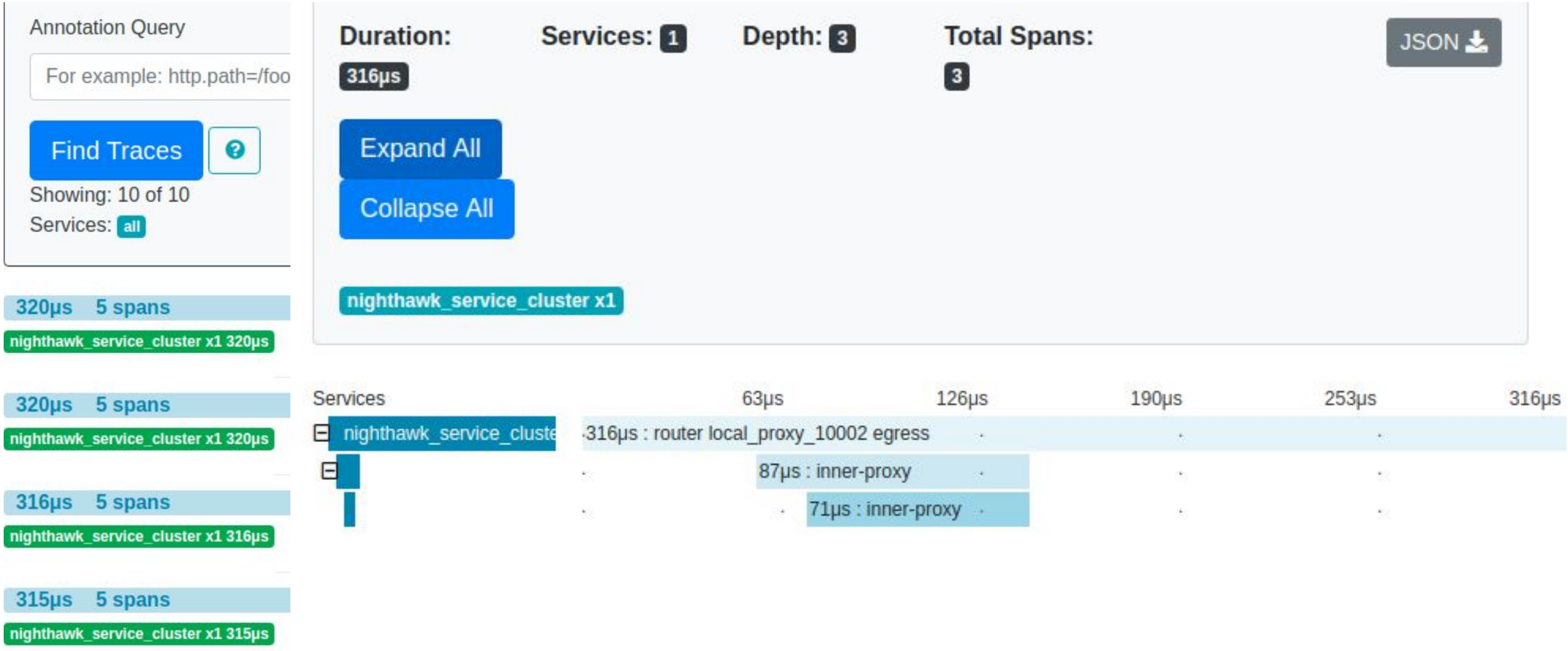
Bad practice: This test (more then) saturated the available computing capacity.
Observed latencies are distorted and inflated!
Max RPS, small request/reply - Envoy on 60 cores, 120 clients (Nighthawk threads) on 20 cores, single connection.
H1: ~641000 rps, H2 ~463000 rps



Bad Practice!
Capacity Saturated
Observed latencies
Distorted & Inflated

Nighthawk: tracing

Supports Zipkin trace initiation. Leverages Envoy's OpenTracing facilities.



Annotation Query

For example: http.path=/foo

Find Traces

Showing: 10 of 10

Services: all

320μs 5 spans
nighthawk_service_cluster x1 320μs

320μs 5 spans
nighthawk_service_cluster x1 320μs

316μs 5 spans
nighthawk_service_cluster x1 316μs

315μs 5 spans
nighthawk_service_cluster x1 315μs

Duration: 316μs

Services: 1

Depth: 3

Total Spans: 3

JSON

Expand All

Collapse All

nighthawk_service_cluster x1

Services | 63μs | 126μs | 190μs | 253μs | 316μs

nighthawk_service_cluste 316μs : router local_proxy_10002 egress

87μs : inner-proxy

71μs : inner-proxy

Nighthawk: notable features

- **Termination predicates:** specify when and how to terminate execution
- **Nighthawk as a service:** bi-directional streaming grpc service to request load tests and receive status updates
- **Replay:** pull to-be-replayed traffic from a grpc service (in review)

Situational use of tools and techniques

- Haphazard
- Performance Annotations
- Microbenchmarks
- Application-level load testing
- **Fuzzing for Performance**

Finding performance problems with fuzzing

- Fuzzers stimulate subsystems with random data
- Finds code-paths missed by unit, integration, and system tests
- Learns what patterns wake up new code and spends more effort varying those
- Security focus, but finds performance issues too

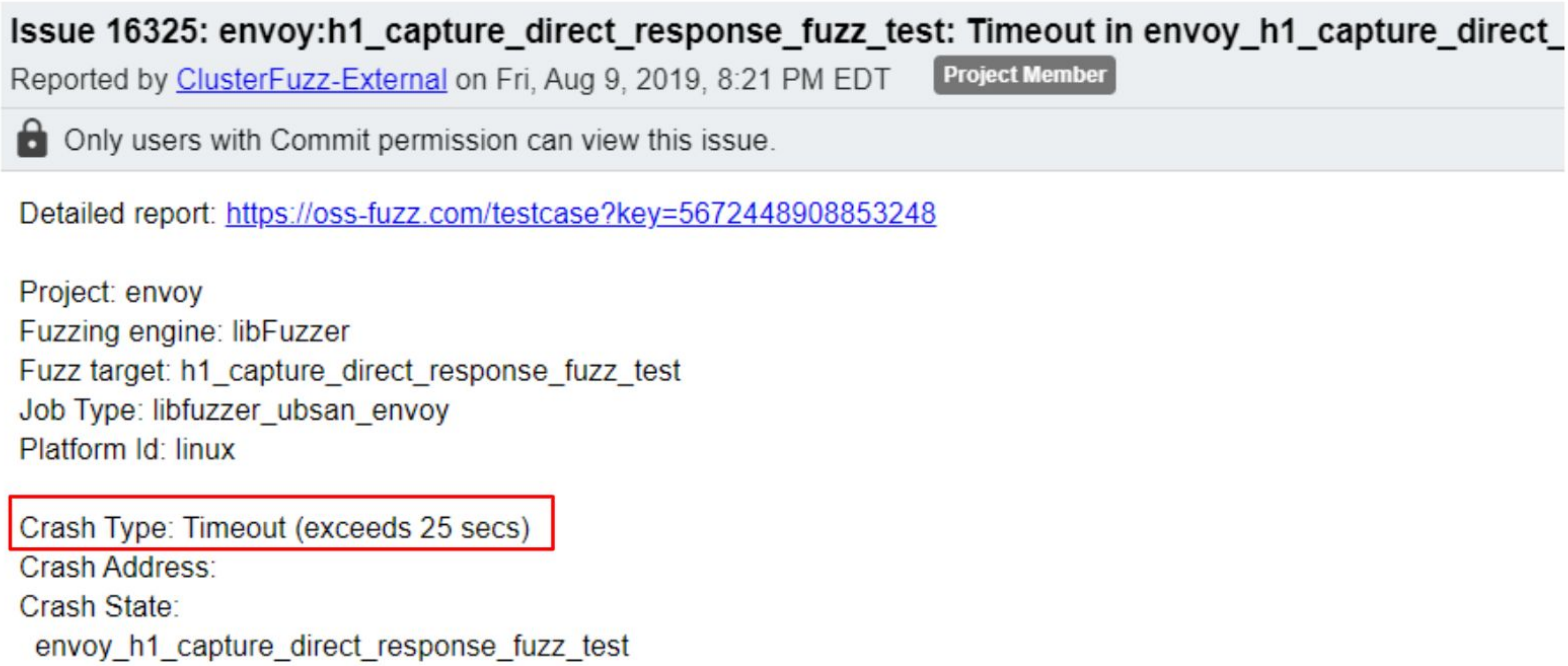
Finding Performance Problems from generated patterns

Fuzzer	Avg Exec/Second	Timeouts (%)	Regular Crash (%)
H1 capture direct response	1.9	3.8	7.3
H1 capture	2.8	1.2	77.7
Access Log Formatter	4.8	26	17.9
Conn Manager	4.8	0.8	22.4
New Buffer	9.3	0	0
Buffer	12.5	0	0

Key Takeaway: Performance data can be hard to find, but low executions/sec, high timeouts may be a signal

https://oss-fuzz.com/fuzzer-stats?group_by=by-fuzzer&date_start=2019-10-01&date_end=2019-10-04&fuzzer=libFuzzer&job=libfuzzer_asan_envoy&project=envoy

Finding Performance Problems from generated patterns



Issue 16325: envoy:h1_capture_direct_response_fuzz_test: Timeout in envoy_h1_capture_direct_

Reported by ClusterFuzz-External on Fri, Aug 9, 2019, 8:21 PM EDT **Project Member**

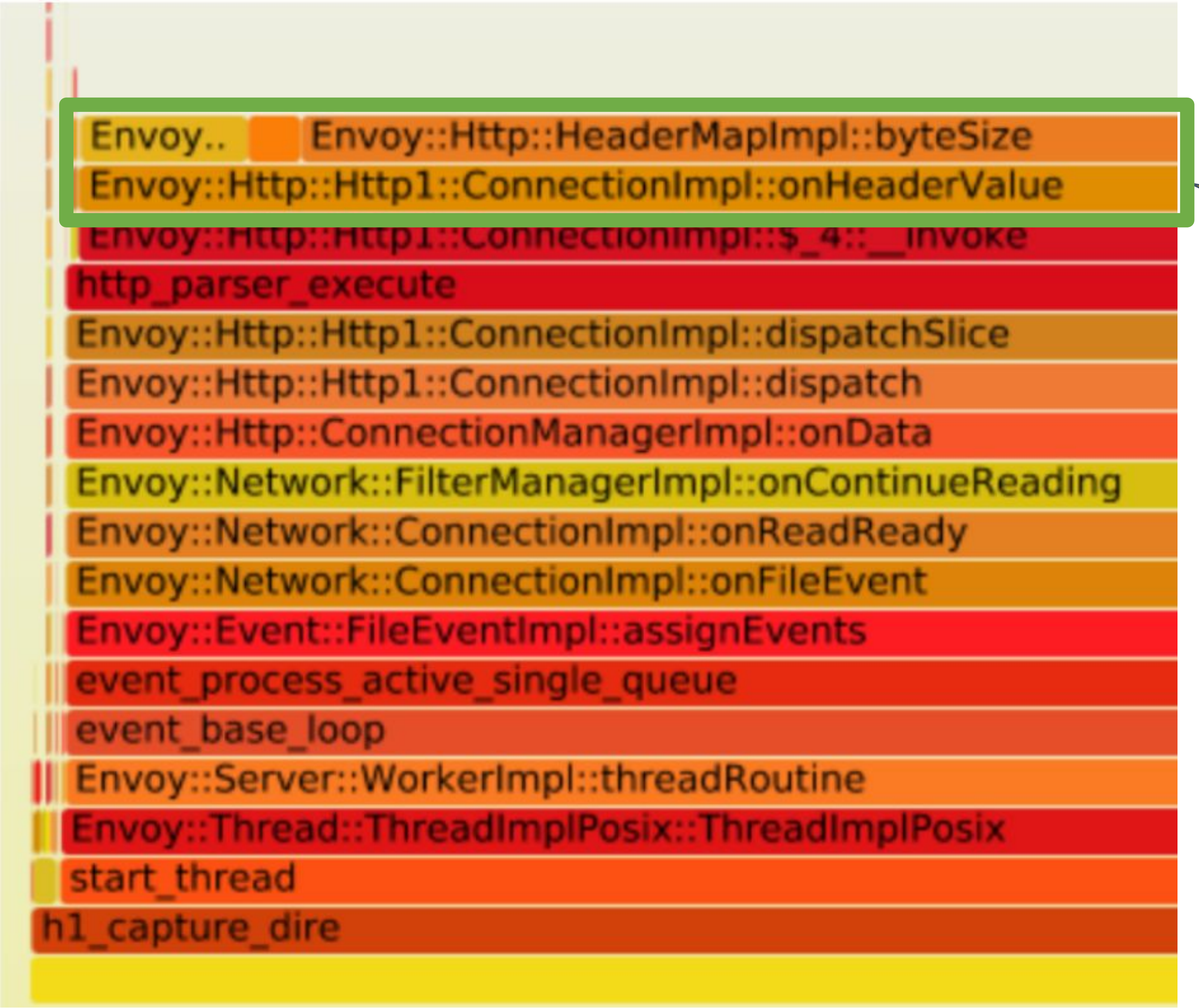
🔒 Only users with Commit permission can view this issue.

Detailed report: https://oss-fuzz.com/testcase?key=5672448908853248

Project: envoy
Fuzzing engine: libFuzzer
Fuzz target: h1_capture_direct_response_fuzz_test
Job Type: libfuzzer_ubsan_envoy
Platform Id: linux

Crash Type: Timeout (exceeds 25 secs)
Crash Address:
Crash State:
  envoy_h1_capture_direct_response_fuzz_test

Flame-Graph from repro of fuzz timeout



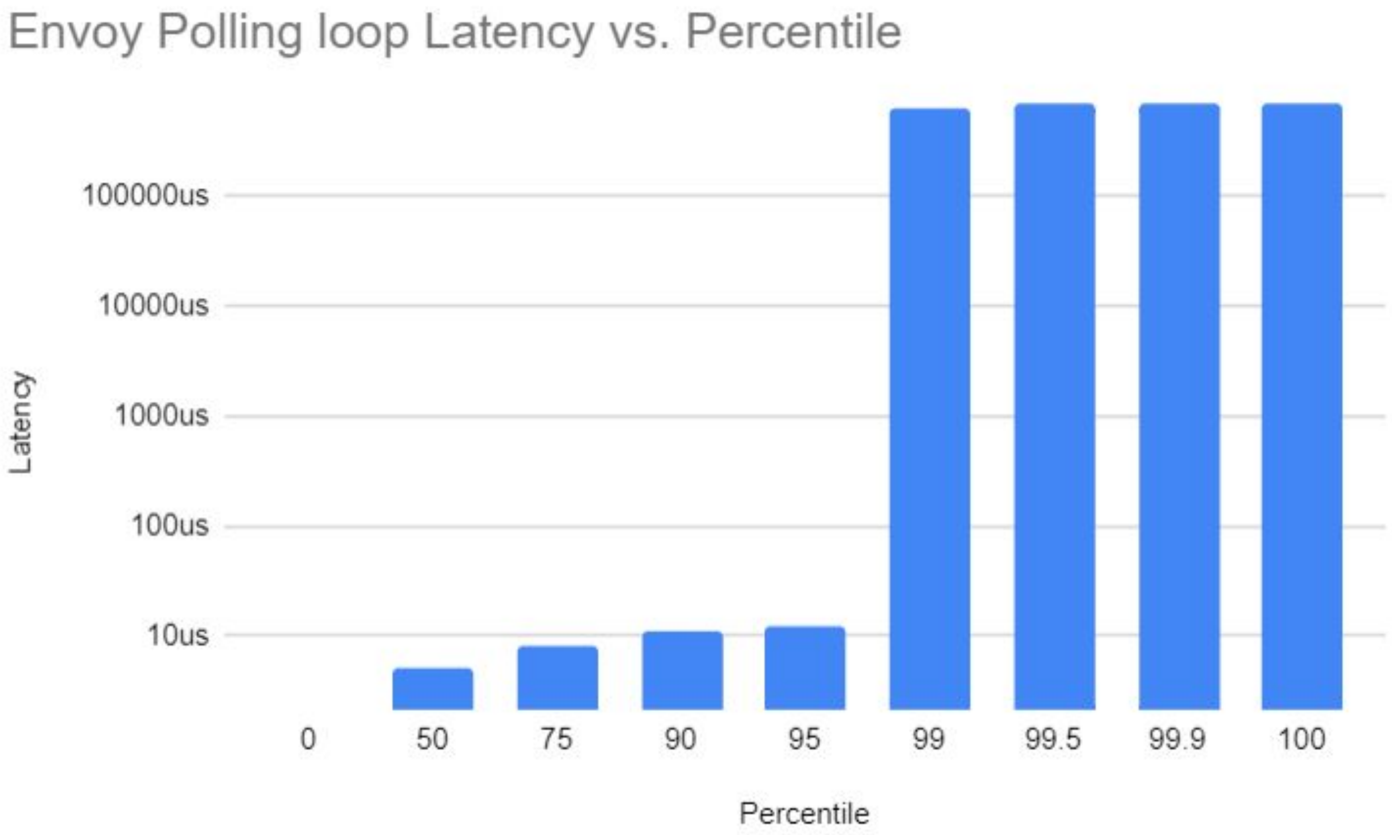
O(n^2) update of byte-size as new headers arrive.

...and the resultant CVE

CVE-2019-15226

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15226

Upon receiving each incoming request header data, Envoy will iterate over existing request headers to verify that the total size of the headers stays below a maximum limit. The implementation in versions 1.10.0 through 1.11.1 for HTTP/1.x traffic and all versions of Envoy for HTTP/2 traffic had **$O(n^2)$ performance** characteristics. **A remote attacker may craft a request** that stays below the maximum request header size but consists of many thousands of small headers to **consume CPU and result in a denial-of-service attack.**



Fuzzing performance challenges

- Most Envoy fuzz tests are very slow, masking problems in production code
- Assertion failures make fuzzing slow or crashy
  - Fuzzing will find more if the fuzzing tests avoid these
- Assertions themselves may be slow
- The tooling is not tuned for this use case at all, but there is opportunity for improvement
- Writing efficient and effective fuzz-tests will help benefit security and performance

Next Steps

- Nighthawk continued development & maturity
  - Continuous performance testing using public clouds
  - Visualization flow improvements
  - Promote as broader HTTP(2) measurement infrastructure
  - Contributors welcome: https://github.com/envoyproxy/nighthawk
- Fuzz testing for performance?
  - Semi-aligned with fuzz-testing for security bugs
  - Value increases as we make faster fuzzers
- "**Fast is Better than Slow!**"

Thank You