

One Contract, Three Runtimes: A Steady-State and Chaos Evaluation of a Social Feed Gateway on Cloudflare Workers, Go, and Spring Boot

J. Michael Duhart

Version 1.0, 2026-10-03. Sections 1 to 3 were written on 2026-10-01, before any run, and are kept as written (the pre-registered draft is commit 28b39d10b in the repository). Where the study departed from them, Section 3.10 says so. Every number in Sections 4 to 7 is computed by the R analysis in `analysis/R/` from the raw harness output in `results/`; the generated tables are `analysis/out/RESULTS.md`.

Abstract

We compare three implementations of one production service, the hot path of a social feed gateway, built to a single written contract: the production TypeScript on Cloudflare Workers (run locally on workerd with D1), a Go port on Postgres, and a Spring Boot port on Postgres. Each ran on one host under hard CPU and memory caps (two 1.5-core replicas behind HAProxy), with an open-model load generator, a step ramp to saturation, six injected faults at 70% of each contender's own saturation, and a separate arm on Cloudflare's network. Single runs were completed for 30 local configurations and 4 edge configurations; 27 planned configurations were not run. Under a 500 ms p99 and 1% error objective, Go sustained 80 arrivals per second, Spring 90 and local Workers 15. In all three the bottleneck was the data tier (Postgres at 96 to 98% of its 2-core cap; the single D1 thread at one full core), while the gateways used 17 to 37% of theirs, so the language runtime did not decide throughput at these caps. It did decide cost and recovery: Go served 138 API requests per gateway core-second against Spring's 106 and Workers' 35, and Spring took 17 to 18 s to recover from a replica kill or restart (an 8 s JVM cold start) against Go's 0.4 to 0.6 s. A per-replica cache raised Go's saturation from 80 to 190, until a 1-core object store became the ceiling, and left a cold-cache tail after a kill (RTO 84 s). The production schema full-scanned the feed: with the production indexes Go saturated at 4 arrivals per second and neither other contender met the objective at all; two indexes, no code change, moved Go to 80. On Cloudflare's edge, the Cache API lowered median latency by 37% at 35 arrivals per second and moved the error knee from 60 to 80 arrivals per second, but D1 capped the edge Worker below the local Postgres contenders. H1 (Workers better under load because of edge caching) is partly supported; H2 holds for graceful termination and CPU theft in most cases and fails on latency during replica kills and database faults; H0 cannot be rejected, because run-to-run variance was not measured. The study's most reusable finding is methodological: the fault-injection proxy, capped at 0.5 CPU, silently set the ceiling of every early Go and Spring run until its own CPU was checked.

1. Introduction

Language and runtime comparisons for web services are usually run on toy endpoints: a "hello world" handler, or a single SELECT. Those benchmarks measure the HTTP stack and the driver and say little about a service whose cost lives in authorization logic, multi-query page assembly, and transactional writes. They also stop at throughput. A production team choosing a runtime needs to know how it behaves when a replica dies, a CPU is stolen, or the database slows down while users are active.

This paper compares three implementations of the same production service, the hot path of the GlobalFeedGateway (the social feed behind four consumer products), built to one written contract:

1. **Workers:** the production TypeScript, unmodified, on workerd with D1 (SQLite).
2. **Go:** a port in Go 1.24 on net/http and pgx.
3. **Spring:** a port in Java 21 on Spring Boot 3 with virtual threads, built with Maven.

It follows the chaos engineering method of Basiri et al. (Netflix, 2016): define steady state from measurable output, hypothesise that steady state holds in the control and the experiment, inject real-world faults, and look for a difference.

Research questions.

- RQ1. Under the same CPU and memory caps, which implementation sustains the highest request rate within a latency objective, and at what resource cost?
- RQ2. During a fault, how far does each depart from steady state, and how long does it take to return?
- RQ3. Which differences come from the language runtime, and which from the surrounding system (pool, load balancer, storage engine, cache, edge)?

Hypotheses. Stated before any run, and tested as stated.

- **H1 (the owner's hypothesis).** Under load, the Workers implementation performs better than Go and Spring because of edge caching: higher throughput within the SLO, and lower p95 and p99 latency.
- **H2 (resilience, per experiment).** Each implementation keeps steady state through each fault within the SLO, and returns to baseline within 30 s after the fault is lifted.
- **H0 (null).** Within the same caps, there is no difference between implementations larger than the spread between repeated runs.

A finding from reading the production code shapes how H1 is tested. The gateway marks every JSON response on the hot path `Cache-Control: private, no-store (src/errors.ts)`. Only share cards, profile headers and media bytes are cacheable. As built, no feed, item or comment response is ever served from the edge cache, and a signed-in feed is personal, so it cannot be shared from a cache at all. H1 is therefore split into three arms:

Arm	Cache	Where Workers runs	What it isolates
A nocache	none (as built)	local workerd, capped	the runtimes alone
B cache	each runtime's own production cache under one policy (<code>contract/CACHING.md</code>): Workers the Cloudflare Cache API, Go an in-process cache with <code>singleflight</code> , Spring Caffeine through Spring Cache	local workerd, capped	how well each ecosystem's native caching converts a shared policy into latency and throughput
C edge	the Workers Cache API on Cloudflare's network	Cloudflare's network, a separate bench Worker with its own D1 and R2	the platform: per-PoP edge cache, elastic isolates, D1 as a service

The cache policy fixes what is cached (shared item rows, media slots, profiles, the suggested-feed candidates, anonymous responses, authorized media bytes), the TTLs (15 s to 300 s), and the invalidation, identically for all three. Viewer-relative data is never cached, and the visibility projection runs on every request whatever the cache returned. Only the mechanism differs, and each is the one a team on that stack would choose.

Arm C has no cap; that is the point of a serverless platform. Its latencies include the public internet from the test host and are not compared directly with arms A and B. The comparison in arm C is the shape of each curve under rising load (where p99 bends, where errors start), not absolute milliseconds.

2. System under test

2.1 The service

The gateway serves a mode-scoped social feed (profiles, posts, reactions, comments). Every read passes a six-clause visibility projection (AX-041) in two forms that must agree: an SQL predicate for lists and a row check for single reads. Hidden items return a 404 that is byte-identical to a nonexistent one. The feed page is assembled from three independently budgeted sections (own, followed, suggested). A section that overruns its 400 ms budget is dropped and named in degraded, and the page still returns. That degradation behaviour is observed directly in the chaos experiments.

2.2 The contract

contract/CONTRACT.md fixes nine routes, the schema (contract/schema.sql), the wire format, the error table, the cursor scheme, the transaction boundaries of every write, and the operational behaviour: readiness, graceful drain on SIGTERM, per-request deadlines carried to the database, bounded pool waits that fail fast with a 503. Each port carries the projection agreement test and its clause mutation test. A port that fails them is not benchmarked.

2.3 Implementations

	Workers	Go	Spring
Language	TypeScript	Go 1.24	Java 21
HTTP	workerd	net/http	Tomcat, virtual threads
DB access	D1 binding	pgx/v5 pool	JDBC, HikariCP
Storage	D1 (SQLite)	Postgres 16	Postgres 16
Concurrency model	isolate event loop	goroutines	virtual threads

3. Method

3.1 Testbed

One host: 16 vCPU, 31 GB RAM, Linux 5.15, Docker. The host also runs unrelated production services. To keep those services stable and to make the comparison fair, every component runs in a container with hard cgroup limits, and the run aborts if host available memory falls below 1.5 GB.

Component	Count	CPU	Memory
Gateway replica	2	1.5 each	1 GB each
HAProxy	1	0.5	128 MB
Postgres 16 (Go, Spring)	1	2.0	1.5 GB
Toxiproxy	1	0.25	default
k6 load generator	1	2.0	1 GB

The system under test is capped at 6.25 of 16 cores. The load generator is capped separately, so it is never starved by the system it measures. Contenders run one at a time, never concurrently.

3.2 Topology

```
k6 (open model) -> HAProxy (roundrobin, /readyz check 1s, fall 2 rise 2, redispatch)
                  | -> replica A --+
```

Workers uses the same front: HAProxy over two workerd processes. Its storage is D1, its production engine. That is a declared confound (Section 6).

3.3 Dataset

A deterministic generator (fixed seed) writes the same logical dataset to Postgres and to D1: about 20,000 profiles, a power-law follow graph of about 400,000 edges, about 200,000 items over 60 days, 300,000 reactions, 150,000 comments, and blocks and mutes. Items cover every visibility state the projection must hide (private, rejected, blocked author, suspended author, subscribers only), so the predicate does real work.

3.4 Workload

Open-model arrival (k6 constant-arrival-rate and ramping-arrival-rate). A closed loop would slow its own request rate when the server slows, which hides latency (coordinated omission). The mix, by request share:

Operation	Share
Feed page	60%
Item read	15%
Profile gallery	8%
Comments read	7%
Reaction put/delete	6%
Comment create	2%
Create + publish	2%

90% of virtual users present a signed RS256 token for a seeded account and 10% are anonymous.

3.5 Steady state and service level objective

Steady state is defined from output, not from internals: p99 latency, error rate (5xx plus timeouts), and achieved throughput, measured per second.

- **SLO:** p99 < 500 ms and error rate < 1%, over a 60 s window.
- **Saturation point (S):** the highest arrival rate in a step ramp (steps of fixed size, 60 s each) at which the SLO holds.
- **Chaos load:** 70% of S for each contender. Each contender is tested at the same relative load, so a fault lands on systems with the same headroom.

Every run starts with a 2-minute warm-up that is discarded, for all three contenders alike (the JVM needs it; the others get it for symmetry).

3.6 Experiments

Each experiment runs a 3-minute control period, the fault, then observation until recovery or 5 minutes. The hypothesis in every case: *steady state holds through the fault, within the SLO, and recovers within 30 s after the fault ends.*

ID	Fault	Injection	What it tests
E1	Instance kill	SIGKILL one replica, restart after 30 s	Failover, health check detection, retry and redispach
E2	Graceful termination	SIGTERM one replica	Readiness flip and connection drain (a deploy)
E3	CPU flood	stress-ng pinned to one replica's CPUs, 60 s	Noisy neighbour, scheduler and GC behaviour under CPU theft
E4	Database latency	Toxiproxy +100 ms $\pm$ 20 ms on the DB link, 60 s	Deadlines, pool saturation, feed section budgets (graceful degradation)
E5	Database outage	Toxiproxy cuts the DB link, 15 s	Fail-fast 503s, no hung connections, recovery

Abort conditions: host available memory < 1.5 GB, or error rate > 50% for 30 s after the fault has ended.

3.7 Metrics

Collected each second, per contender, arm, experiment, run and phase (warm-up, control, fault, recovery), in one tidy long-format table that the R analysis reads.

Metric	Definition	Why it matters
p50, p90, p95 , p99 , p99.9 latency	per-second percentiles from per-request timings	the mean hides the users who are suffering; the tail shows them
Throughput	achieved requests per second against offered	a system that sheds load looks fast if only the survivors are timed
Error rate	5xx plus timeouts over all requests; 4xx reported apart	graceful degradation versus failure
Degraded rate	200 responses that name a skipped feed section	the designed middle ground between full success and an error
Saturation	CPU % and memory % of each container against its cap; HAProxy queue length; DB pool in use	whether the system degrades gracefully or falls over
RTO	fault lifted until throughput is within 5% of control and p95 and p99 are within 10% of control, held for 10 s	how long users feel the fault after it is gone
MTTD	fault start until the load balancer marks the replica down	how long the system routes users into a dead replica
Error budget burn	failed requests over total, against a 1% budget	the SRE view of how much of the month one fault costs
Apdex (T = 100 ms)	(satisfied + tolerating/2) / total	a single user-satisfaction score per phase
Efficiency	requests per core-second, and per GB of memory	what the same traffic costs to serve
Cold start	container start until its first 200	how fast a replacement replica takes traffic
GC and runtime	Go GC pause and goroutines; JVM GC pause, heap, threads	where a tail spike comes from

3.8 Repetitions and statistics

Each configuration runs three times. Reported values are medians across the three runs, with the range. Latency distributions are compared by their full CDF, not only by percentiles. Differences smaller than the spread between runs are reported as no difference.

3.9 Reference: the production edge

As context only, a polite probe (at most 2 requests per second for 5 minutes) records anonymous feed latency from this host to the production Worker on Cloudflare’s network. That number includes the internet and the global edge. It is not comparable to the capped runs and is not ranked against them.

3.10 Deviations from the pre-registered method

Sections 1 to 3.9 above are the plan as written on 2026-10-01. The study departed from it in the following ways. Each is stated here rather than edited into the plan.

1. **Testbed caps.** The fault proxy (Toxiproxy) was planned at 0.25 CPU. It first ran at 0.5 CPU and 128 MB, and at that cap it was the ceiling of every Go and Spring run (Section 6.1). It was raised to 2 CPU and 512 MB, and every Go and Spring run analysed here ran at that cap; its p95 CPU in those runs was at most 37% of the cap. The three Workers runs analysed (ramp, baseline, kill) predate the change and ran with Toxiproxy at 0.5 CPU; its p95 CPU there was at most 33% of that cap, so the cap did not bind. The load generator ran at 2 CPU and **2 GB**, not 1 GB. The table in Section 3.1 also omits components that were present and capped: MinIO (object store for Go and Spring, 1 CPU, 512 MB), Redis (rate limits and idempotency for Go and Spring, 0.5 CPU, 256 MB), and the Workers store host (the single owner of the local D1 and R2, 2 CPU, 1.5 GB). HAProxy ran with retries 1, option redispatch, connect timeout 1 s, server timeout 10 s.
2. **Experiments.** A sixth fault was added: **E6 obj latency**, +150 ms (jitter 30 ms) on the object-store link for 60 s. Two non-fault experiments were added: a **soak** (20 minutes at 70% of saturation) and a **diurnal** run (24 UTC hours of an Instagram-shaped arrival curve compressed to 24 minutes, peak 0.9 x saturation). The chaos timeline was warm-up 120 s, control 120 s, fault, then a fixed 180 s of recovery, not “until recovery or 5 minutes”.
3. **Ramp.** Steps were 10 arrivals per second from 20 (Go, Spring) and 5 from 5 (Workers), 60 s each. The first 5 s of each step are excluded from its statistics. The saturation rule adds a third condition the plan did not state: no dropped arrivals (k6 could start every scheduled iteration). Section 4.3 reports the step-aggregate rule as primary and four other definitions beside it.
4. **Chaos load.** 70% of saturation, rounded down by the harness: Go 56, Spring 63, Workers 10, Go with cache 133 arrivals per second. The edge baseline ran at 35 arrivals per second: the queue script’s fallback (70% of a default of 50), because the edge ramp produced no latency-based saturation point (Section 4.7).
5. **Repetitions.** Three runs per configuration were planned. Only pass 1 completed (Section 6.4). Section 3.8’s medians and ranges therefore cannot be computed; every value is a single run, and no difference is called significant.
6. **Schema dimension.** The smoke runs showed that the production schema full-scans the feed (Section 4.4). A second dimension was added: `--schema tuned`, which adds indexes only (no query or code change) from an index study (contract/tuned/STUDY.md). All main results use the tuned schema; the three production-schema ramps are kept as a finding. They ran with a 60 s warm-up, steps of 2 arrivals per second from 2, and before the Toxiproxy fix (its p95 CPU was at most 22.5% of its 0.5 CPU cap in them).
7. **The edge arm.** The reference probe of Section 3.9 (the production Worker) was not run. Instead, arm C ran against a separate bench Worker with its own D1 (seeded with the same dataset and the tuned indexes), R2 and KV, at `feedbench.designsbyduhart.org`. Its ramp (20 arrivals per second every 60 s from 20 after a 120 s warm-up at 20, owner cap 300) did not record its plan; the step boundaries were reconstructed from the queue script and confirmed against request counts per minute. Both ramps were stopped by k6’s abort rule (more than 50% of all requests failed). In the ramps the anonymous share was lowered from 11.1% to 3.0% of arrivals to stay under Cloudflare’s per-IP limit for anonymous reads.

8. **Workers queue.** The Workers contender ran with the development queue enabled (WORKERS_QUEUE=1), so uploaded images are released and published as in the two ports.
9. **Images rebuilt.** A host-wide prune by an unrelated session removed the gateway images on 2026-10-03 at 06:45 UTC. They were rebuilt from unchanged source. One analysed run (Spring dblatency) ran on the rebuilt Spring image (411a3a935b05); every other Spring run ran on 9b57f616e341. It completed normally eight minutes before the owner stopped the queue and is included, flagged.
10. **Not run.** The Spring and Workers cache arms; Spring dbout, objlatency, diurnal and soak; Workers term, cpu, dblatency, dbout, objlatency, diurnal and soak; repetitions 2 and 3 of everything; and a runtime-bound profile (gateways at 0.5 CPU, database at 4 CPU) whose queue entries failed to start. These are gaps. Nothing is imputed for them.
11. **Metrics.** Go GC and JVM runtime gauges (Section 3.7) were scraped but are not analysed. Efficiency is computed by integrating the cgroup samples over their own timestamps (Section 4.3), because the harness's per-second core series has gaps. MTTD is fault start to the first HAProxy sample (1 s sampling) showing the replica DOWN.

4. Results

All results are single runs on the tuned schema unless stated. Errors are HTTP 5xx plus requests that received no status; 4xx are reported apart (every contender returns 3 to 4% 404s, by design: 15% of item reads ask for hidden or nonexistent items). Latency is of API requests; media byte fetches are reported separately.

4.1 What ran

Thirty local runs and four edge runs are analysed (analysis/out/tables/runs.csv). Go is complete for both arms: ramp, baseline, all six faults, diurnal and soak (no cache) and ramp, baseline and all six faults (cache). Spring has ramp, baseline and four faults (kill, term, cpu, dblatency). Workers has ramp, baseline and kill. Each contender has a production-schema ramp. The edge has a ramp and a 5-minute baseline in each cache mode. 27 planned configurations were not run (Section 3.10, item 10).

4.2 Steady state

Table 1 is the control phase of each 5-minute baseline. Each contender ran at 70% of its own saturation, so the rows are equal in headroom, not in load.

Table 1. Baselines (API requests, control phase).

Contender	Offered (arrivals/s)	API rps	p50 (ms)	p95 (ms)	p99 (ms)	p99.9 (ms)	Errors	Apdex (100 ms)	Media p95 (ms)
Go	56	59.0	21.6	44.3	59.4	88.6	0.00%	1.000	10.1
Go + cache	133	127.9	8.9	31.2	1,798.4	5,002.9	0.92%	0.982	5,001.5
Spring	63	66.2	22.7	47.8	68.1	105.6	0.00%	0.999	13.3
Workers (local)	10	10.5	39.6	108.1	196.7	333.7	0.00%	0.967	38.8
Edge, Cache API on	35	36.8	772.8	1,339.9	1,644.3	1,993.9	0.00%	0.041	560.0
Edge, no cache	35	36.8	1,237.2	1,907.3	2,350.6	3,036.2	0.00%	0.002	718.5

Go and Spring are close at steady state: p99 59 and 68 ms. Workers, at a sixth of their load, has three times their p99. The Go cache arm has the lowest median (8.9 ms) and the worst tail of any local baseline. For about the first minute of control its p99 was 40 to 66 ms per 20 s window; then the object store saturated (Section 4.5), media fetches began to time out at 5 s, uploads drew 503s, and the p99 per 20 s window rose to between 1.6 and 5.0 s. Figure 1 shows the distributions.

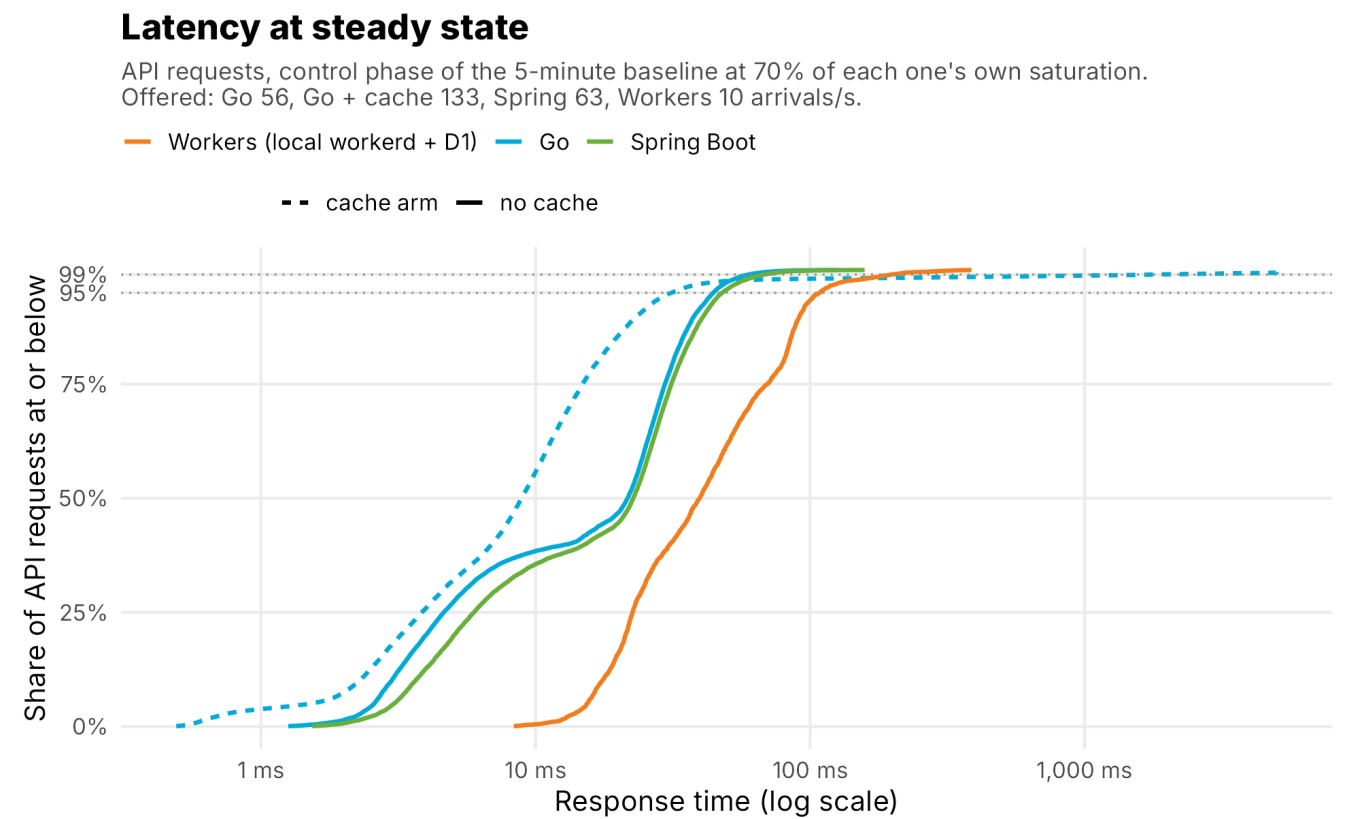


Figure 1. Latency CDFs at steady state (log scale). Each at 70% of its own saturation.

At equal offered load the ramp steps give a like-for-like view (equal_load_latency.csv). At 50 arrivals per second Go and Spring have the same step p99 (50.2 and 50.1 ms) and Go with its cache 29.7 ms. At 80, Go 57.7, Spring 70.4, Go with cache 43.1 ms. Workers, at 20 arrivals per second, already has a step p99 of 2,151 ms.

4.3 RQ1: throughput within the objective, and its cost

Table 2. Saturation (arrivals per second) under five definitions.

Contender	Step aggregate (primary)	Knee	Without the dropped-arrival rule	Every second	Errors only	Harness (Python)	Busiest container at the first failing step
Go	80	80	80	80	80	80	Postgres 96.4% (gateways 18.0%) at 90
Go + cache	190	170	190	190	200	190	MinIO 84.7% (Postgres 75.1%, gateways 32.9%) at 180
Spring	90	90	90	90	90	90	Postgres 97.8% (gateways 37.3%) at 100
Workers (local)	15	15	15	10	25	15	D1 store host 49.4% of 2 cores, one full core (gateways 17.4%) at 20

Definitions (all over 60 s steps, first 5 s excluded): *step aggregate*, the highest step whose p99 over the step is under 500 ms, errors under 1% and no arrival dropped; *knee*, the last step before the first step that fails that rule; *every second*, the original R rule, every per-second p99 in the step under 500 ms; *errors only*, errors under 1% with latency ignored. CPU is the mean over the step as a percentage of the container’s own cap.

The definitions agree for Go and Spring. They disagree in two informative places. Workers is 15 by the step rule and 10 by the per-second rule: at 15 arrivals per second the step p99 is 323 ms, but one second inside the step reached 560 ms. The Go cache arm failed at 180 (40 dropped arrivals and a 251 ms p99) and passed again at 190, so the highest passing step (190) and the knee (170) differ. R’s primary rule reproduces the harness’s Python rule exactly in all nine ramps.

The data tier set every ceiling. At the first failing step Postgres was at 96 to 98% of its 2-core cap for both ports, while the gateway replicas averaged 18% (Go) and 37% (Spring) of their 1.5-core caps. The Workers store host, which runs D1’s single SQLite thread, sat at 49.4% of 2 cores, which is one core fully used, while the Workers replicas used 17%. None of the three gateways was the bottleneck. At these caps RQ1’s throughput question is answered by the storage engine, not the runtime: Postgres with a pool of 32 per replica against one SQLite writer thread.

Where each runtime bends

Step ramp, 60 s steps, tuned indexes. Step-aggregate p99 (first 5 s of each step skipped).

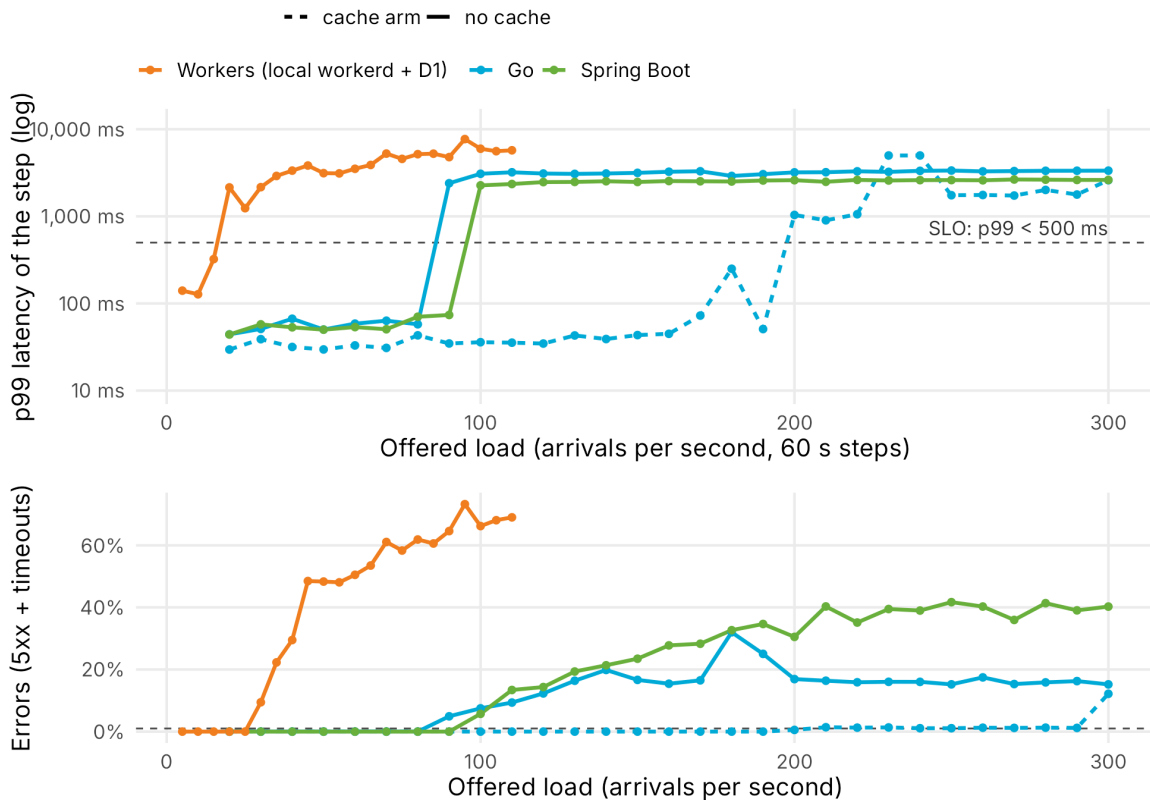


Figure 2. Step ramp: p99 per 60 s step (top, log scale) and errors (bottom). Tuned indexes.

The cost of a request does differ by runtime (Table 3). In the baseline control phase Go served 138 API requests per gateway core-second, Spring 106 and Workers 35. The cache arm raised Go to 193, because a cache hit costs less CPU than a database round trip. Memory differs by an order of magnitude: the two Go replicas together held 28 MB, the two Spring replicas 594 MB and the two workerd replicas 1.37 GB.

Table 3. Efficiency, baseline control phase (gateway replicas only).

Contender	Mean gateway cores	API req per gateway core-second	All req (API + media) per gateway core-second	Gateway memory (both replicas)
Go	0.428	137.8	426.2	0.028 GB
Go + cache	0.660	193.0	564.7	0.884 GB
Spring	0.625	106.1	327.9	0.594 GB
Workers (local)	0.304	34.6	111.0	1.368 GB

The planned efficiency table first reported NA for Spring. The cause was in the analysis, not the data: the harness's per-second `gateway_cores` series has gaps where the 1 Hz cgroup sampler missed a second, and a sum without `na.rm` propagated the gap. The table above integrates the cgroup samples over their own timestamps, for all three the same way.

4.4 The production schema full-scans the feed

With the production indexes, Go met the objective only up to 4 arrivals per second (p99 430 ms at 2), with Postgres at 99.9% of its cap at 6. Spring never met it: at 2 arrivals per second its p99 was 10.0 s and 19.4% of requests failed. The Workers ramp never reached its first measured step: 70.7% of requests failed during the 2-arrivals-per-second warm-up

and k6’s abort rule stopped the run. With two indexes (Go and Spring: one on Postgres; Workers: two on D1), and no query or code change, saturation rose to 80, 90 and 15 (Figure 3).

The cause, from the index study (contract/tuned/STUDY.md, measured separately from this pipeline): every feed section’s WHERE clause is `mode AND audience IN (...) AND archived_at IS NULL AND (ownerBranch OR visitorBranch)`, and the existing partial indexes require `state = 'ready'` or `visibility = 'public'`, which that predicate does not imply. SQLite uses a partial index only when the query implies its WHERE, so on D1 the planner read the whole table and sorted (own, followed and suggested sections at 0.40, 0.47 and 1.35 s median per query). On Postgres the suggested section took 1.6 to 1.8 s on a generic plan. The production Billboard feed has the same predicate shape; the finding was reported to the owner and production was not changed by this study.

The production schema full-scans the feed

Saturation (highest 60 s step with p99 < 500 ms, errors < 1%, no dropped arrivals), no cache, same data.

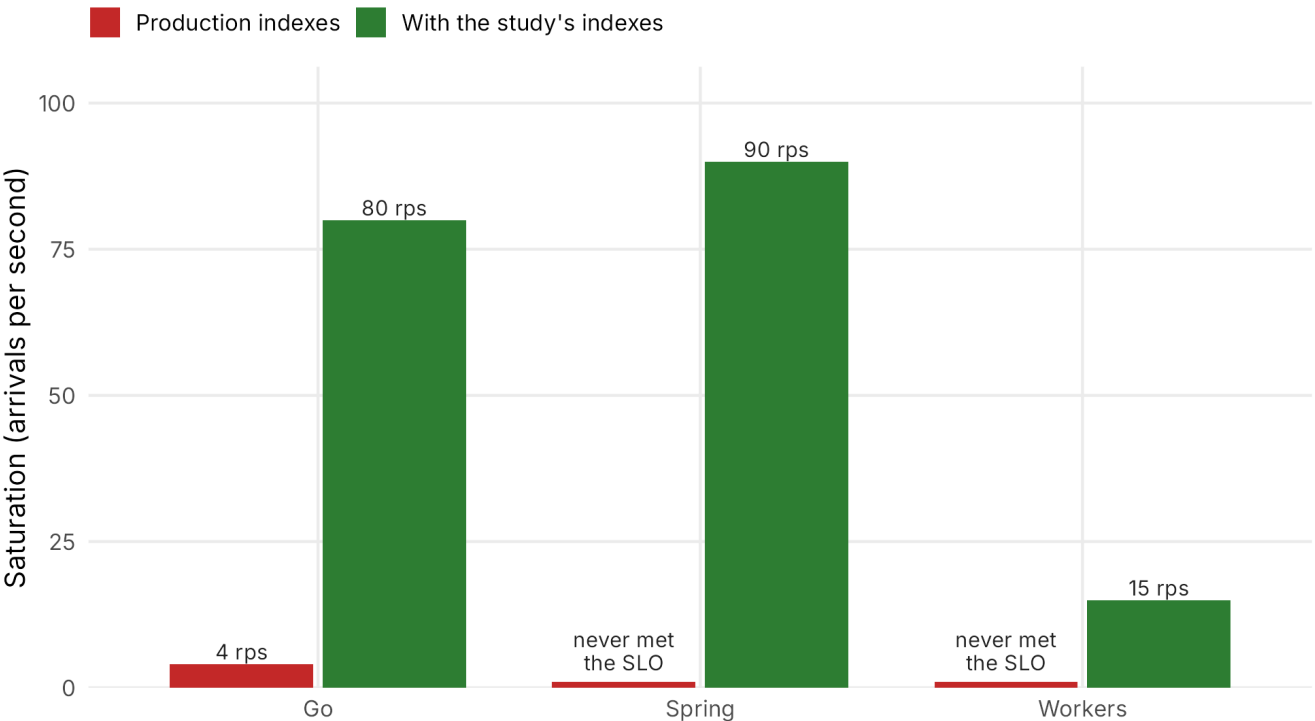


Figure 3. Saturation with the production indexes and with the study’s indexes. Same data, no cache.

4.5 The cache arm (Go)

Go’s in-process cache (singleflight, 256 MiB per replica, the shared policy of contract/CACHING.md) raised saturation from 80 to 190 arrivals per second and lowered the step p99 at every common load (Section 4.2). Hit ratios in the baseline control phase: shared media slots (K2) 98.0%, suggested-feed candidates (K4) 97.0%, anonymous responses (K5) 57.5%, media bytes (K6) 58.2%, profiles (K3) 41.4%, item rows (K1) 3.9%. The item-row cache barely hits: 30 s TTL, per replica, over 200,000 items read at random.

The cache moved the ceiling to the next component. In the ramp, MinIO (the object store, 1 CPU) was the busiest container at every step from 20 upward and reached 99.9% at 200; its cgroup samples are absent from the 230 step onward, which is consistent with the container exiting, though we did not confirm the cause. In the 5-minute baseline at 133 arrivals per second it reached 100% CPU and 100% of its memory cap about a minute into the control phase, the gateways logged S3 SlowDown (“Please reduce your request rate”), and the baseline’s tail followed (Table 1). The cache

arm's ceiling on this testbed is therefore the object store, not the gateway or the database, and its 190 is a 60 s-step figure that the same system did not sustain for 5 minutes at 70% of it.

4.6 RQ2: faults

Table 4 lists every fault run. MTTD is fault start to the first 1 s HAProxy sample showing a replica DOWN; *re-admit* is fault end to the first sample showing it UP again. RTO is fault end to the start of the first 10 s window whose throughput is within 5% of control and whose p95 and p99 are within 10% of control. R and the harness's Python agree on every RTO and MTTD to the millisecond (`chaos_crosscheck_python.csv`).

Table 4. Faults (one run each).

Contender	Fault	MTTD (s)	Re-admit (s)	RTO (s)	Control p99 (ms)	Fault p95 (ms)	Fault p99 (ms)	Fault errors	Fault budget burn	Restart cold start (ms)
Go	kill	3.8	2.3	0.4	48.8	1,006.1	1,034.1	0.00%	0.0	1,523
Go	term	1.7	1.4	0.6	67.2	43.7	57.0	0.00%	0.0	667
Go	cpu	none		50.6	46.8	45.8	61.8	0.00%	0.0	
Go	dblatency	none		0.9	56.6	1,130.6	1,500.2	0.00%	0.0	
Go	dbout	1.8	1.8	0.9	54.4	1.4	1.6	99.52%	99.5	
Go	objlatency	none		0.8	49.0	54.7	777.6	0.00%	0.0	
Go + cache	kill	4.5	1.9	83.8	42.3	1,006.7	1,038.9	0.00%	0.0	1,186
Go + cache	term	1.8	1.5	1.4	39.2	26.7	55.6	0.00%	0.0	735
Go + cache	cpu	none		0.9	54.5	28.6	46.3	0.00%	0.0	
Go + cache	dblatency	9.0	1.0	20.7	42.3	2,636.2	3,190.6	12.52%	12.5	
Go + cache	dbout	1.4	1.4	0.5	45.7	1.3	2.5	97.64%	97.6	
Go + cache	objlatency	none		0.4	46.7	40.1	787.9	0.00%	0.0	
Spring	kill	3.8	9.2	18.3	53.9	1,010.5	1,077.2	0.00%	0.0	8,003
Spring	term	1.9	9.6	16.6	70.9	39.3	57.5	0.00%	0.0	7,997
Spring	cpu	none		8.1	52.5	54.3	80.2	0.00%	0.0	
Spring	dblatency	9.3	2.3	2.5	60.7	2,974.7	3,912.7	41.57%	41.6	
Workers (local)	kill	4.2	3.4	17.9	212.6	1,056.0	1,175.9	0.00%	0.0	1,444

“none” means HAProxy never marked a replica down during the fault. Budget burn is the error rate in the fault window over the 1% budget (99.5 means the window burned errors at 99.5 times the sustainable rate).

Time to recover after the fault is lifted (RTO)

Throughput within 5% and p95/p99 within 10% of control, held for a rolling 10 s window. One run each.

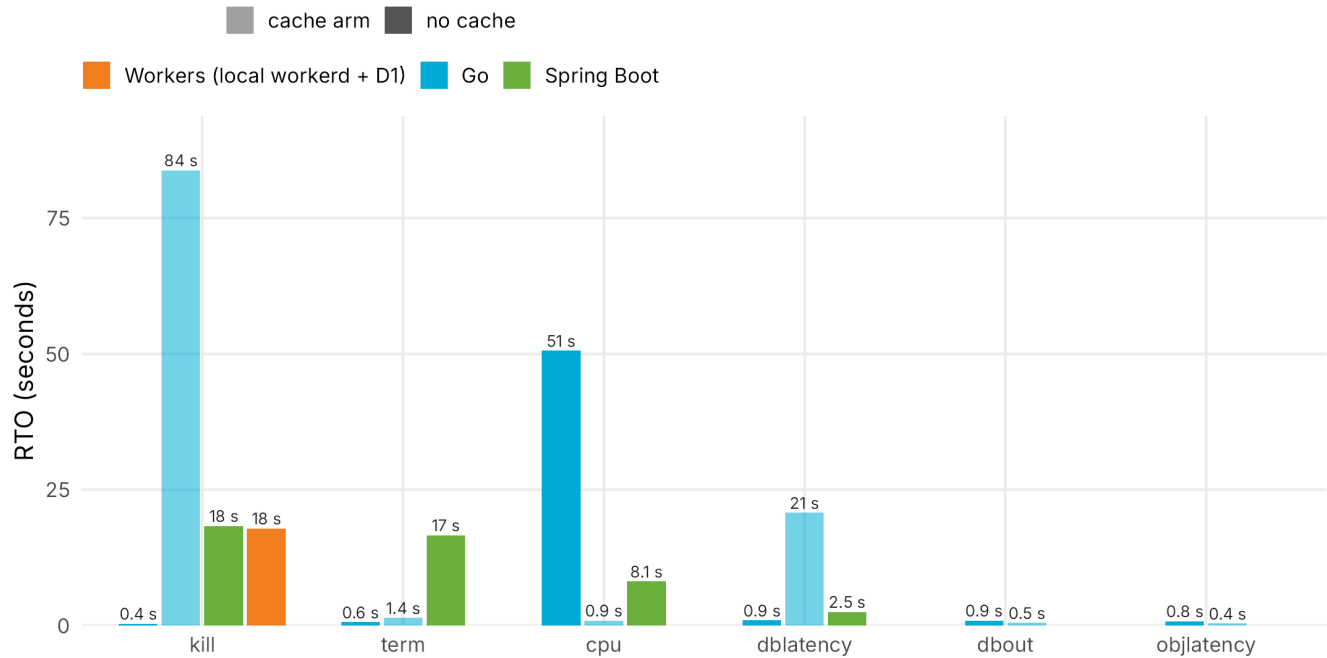


Figure 4. RTO by fault and contender. Bars at zero height with a label are sub-second.

E1 kill. No contender returned an error: HAProxy's retries 1 and redispach re-sent the requests that hit the dead replica. Latency did suffer. In the 3.8 to 4.5 s before the replica was marked down, requests routed to it waited out the 1 s connect timeout before being redispached, so every contender's fault-window p95 and p99 sit just above 1 s. Recovery is where the runtimes separate. Go was back to control within 0.4 s of the restart (cold start 1.5 s). Spring needed 18.3 s: the JVM took 8.0 s to answer its first readiness probe, HAProxy re-admitted it 9.2 s after the restart, and the first minutes on a cold JIT stayed above the 10% band (Figure 5). Workers took 17.9 s with a 1.4 s cold start; its p99 was back in the band at 6.9 s and its throughput later. The Go cache arm took 83.8 s, the longest recovery in the study: the restarted replica came back with an empty in-process cache, and p99 stayed above 110% of its 42 ms control for 68 s while it refilled (Figure 6). The absolute excess is tens of milliseconds; it is the RTO rule's 10% band that makes it a long recovery.

Fault kill: p99 and errors, aligned to fault start

5 s bins of API requests; the shaded band is the fault window.

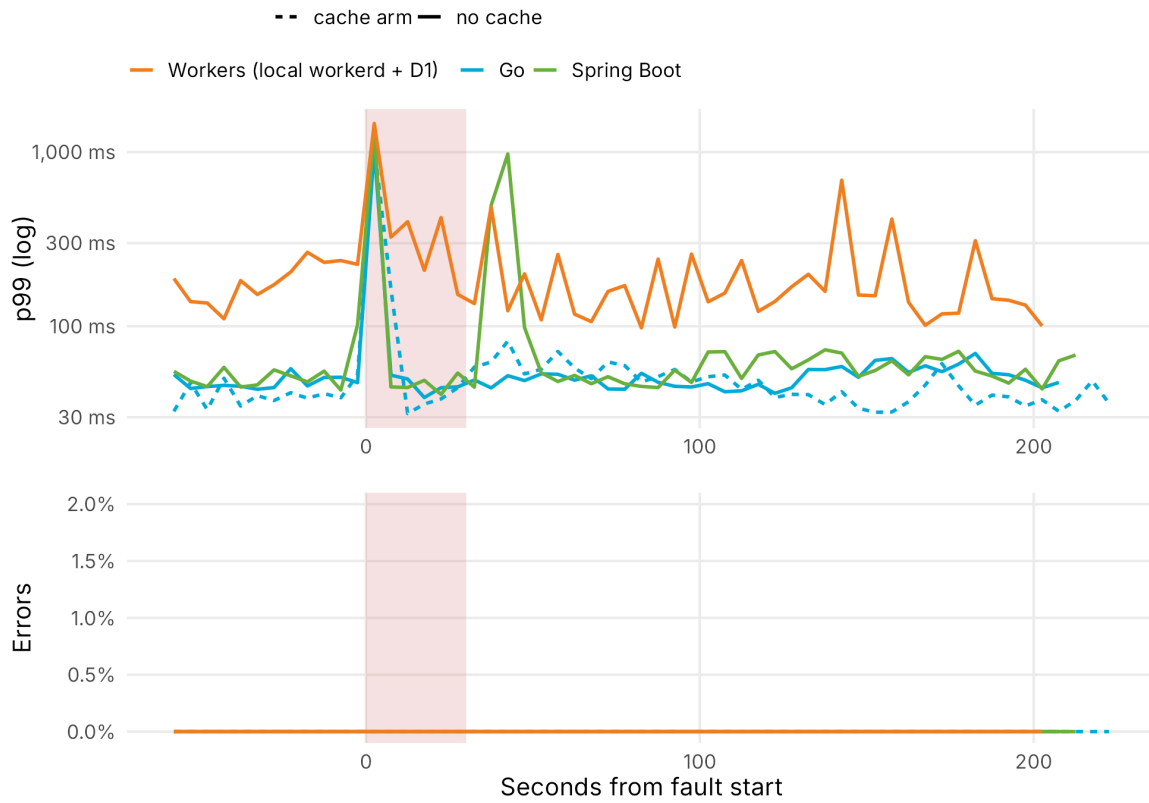


Figure 5. E1 kill: p99 and errors in 5 s bins, aligned to fault start.

Go, one replica killed: the cache arm's cold tail

The restarted replica rejoins with an empty in-process cache; p99 stays above control for over a minute.

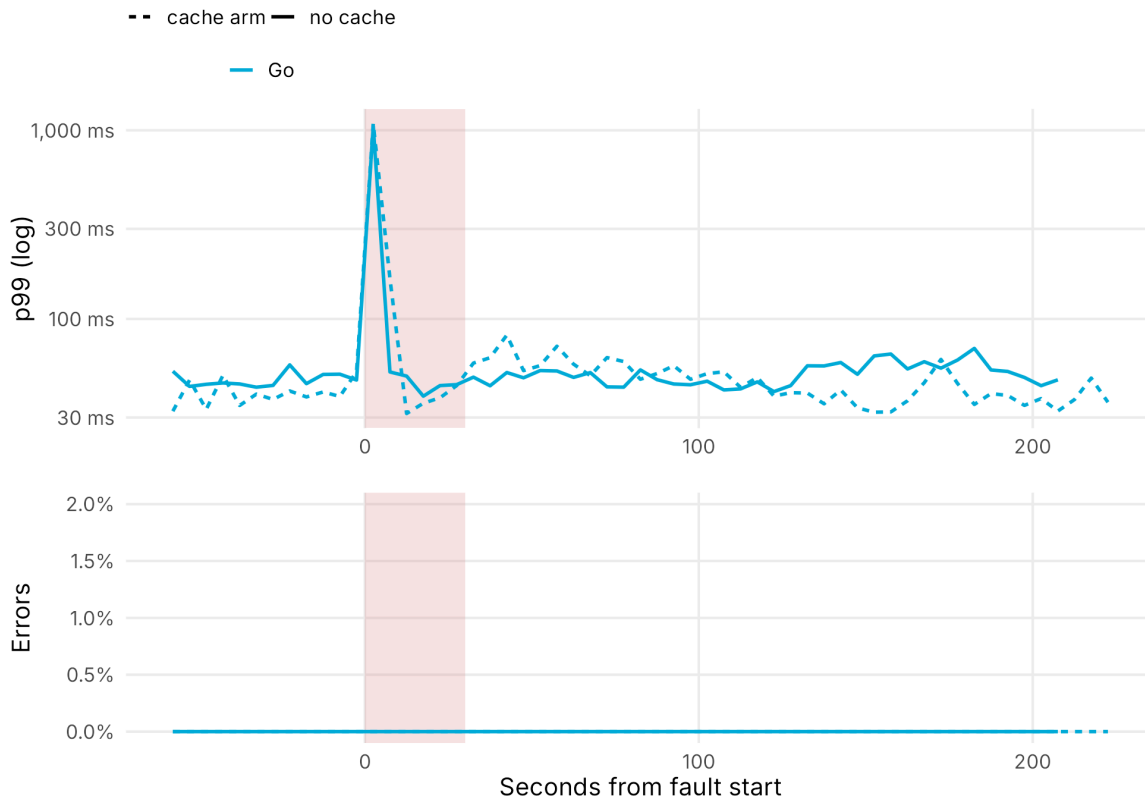


Figure 6. Go, one replica killed: no cache against the cache arm.

E2 term. The graceful path worked as designed for both measured contenders: readiness flipped, HAProxy marked the replica down in 1.7 to 1.9 s (inside the 2.5 s shutdown delay), and no request failed. Fault-window p99 stayed at control level (55.6 to 57.5 ms). Spring's recovery again carries the JVM: 16.6 s against Go's 0.6 and 1.4 s.

E3 cpu. A stress-ng co-tenant pinned to one replica's CPUs did not trip readiness in any run, as the hypothesis predicted (a SELECT 1 stays cheap), so HAProxy never routed around it. The fault-window p99 stayed under 81 ms everywhere. Go without cache logged a 50.6 s RTO with no visible fault-window damage. Its p99 met the band at 27.6 s and throughput met it later. In a single run we cannot separate that from noise at a 10% band, and we do not interpret it.

E4 dblatency (+100 ms on the database link). This fault separates the contenders most. Go without cache absorbed it with no errors at 56 arrivals per second: p99 rose to 1.5 s, 62 responses dropped a feed section and said so in degraded, and HAProxy never marked a replica down. Go with its cache, at 133 arrivals per second, failed 12.5% of requests during the fault. Spring, at 63, failed 41.6%, and HAProxy marked a replica down at 9.3 s. Each request holds a pooled connection for every statement it runs, and +100 ms per statement multiplies the time a feed page holds one; with 32 connections per replica, Little's law puts the limit near the offered load. The runs differ in offered load, so this table shows where each configuration broke at its own operating point, not which runtime is more robust to database latency.

E5 dbout (15 s cut). Go failed fast, as the contract requires: 99.5% (no cache) and 97.6% (cache) of requests in the window failed, with a p99 of 1.6 and 2.5 ms, because a broken connection answers immediately with 503 upstream_unavailable rather than hanging. Readiness failed in turn and HAProxy marked a replica down in 1.4 to 1.8 s. Recovery after the link came back was under 1 s. Not run for Spring or Workers.

E6 objlatency (+150 ms on the object store). API latency moved only at the tail (p99 778 and 788 ms, from uploads that write to the object store) and recovered in under 1 s. Not run for Spring or Workers.

Table 5. H2 per fault and contender. H2 holds when the fault window meets the SLO (p99 under 500 ms, errors under 1%) and the RTO is at most 30 s.

Fault	Go	Go + cache	Spring	Workers (local)
kill	fails (p99 1,034 ms; RTO 0.4 s)	fails (p99 1,039 ms; RTO 83.8 s)	fails (p99 1,077 ms; RTO 18.3 s)	fails (p99 1,176 ms; RTO 17.9 s)
term	holds	holds	holds (RTO 16.6 s)	not run
cpu	fails on recovery (RTO 50.6 s)	holds	holds	not run
dblatency	fails (p99 1,500 ms; 0 errors)	fails (12.5% errors)	fails (41.6% errors)	not run
dbout	fails (99.5% errors, by design)	fails (97.6% errors, by design)	not run	not run
objlatency	fails (p99 778 ms)	fails (p99 788 ms)	not run	not run

Saturation during the faults (`saturation_during_fault.csv`): the surviving replica in E1 rose to 60% (Go), 79% (Spring) and 34% (Workers) of its CPU cap at peak, so the survivor had headroom in every case. Postgres peaked at 73 to 99% in the kill runs, MinIO at 94 to 100%, and HAProxy at 101% of its 0.5 CPU cap in the Go cache kill run.

4.7 The edge arm and H1

On Cloudflare’s network the bench Worker never met the 500 ms p99 objective, even at 20 arrivals per second (p99 877 ms with the Cache API, 910 ms without), because every request from this host crosses the public internet to the nearest data centre and every D1 query travels from there to the database. The edge is therefore compared by the shape of its curves and its error knee (Figure 7, Table 6).

Table 6. Edge ramp (Workers on Cloudflare, tuned D1).

Arrivals/s	Cache API: p95 of successes (ms)	Cache API: errors	No cache: p95 of successes (ms)	No cache: errors
20	713	0.00%	716	0.00%
40	2,436	0.00%	3,024	0.00%
60	6,634	0.49%	6,029	14.17%
80	5,973	44.17%	4,895	49.20%
100	5,657	54.52%	4,703	55.21%

The Cache API helped at every load where both arms were healthy. At 35 arrivals per second for 5 minutes it cut the median from 1,237 to 773 ms (37%) and p99 from 2,351 to 1,644 ms; the bootstrap interval for the p99 ratio, 0.70 [0.68, 0.73], describes sampling noise only (Section 4.9). It moved the first step with errors at or above 1% from 60 to 80 arrivals per second. The errors were HTTP 500s returned by the Worker, most likely D1 overload, which we did not confirm from Cloudflare’s side. Hit ratios at the edge in the cache baseline were 98.9% (K1), 93.5% (K2), 96.4% (K4), 62.9% (K5) and 24.9% (K3), counted from the x-cache header and so not directly comparable with Go’s counter-based ratios.

Workers on Cloudflare's edge: where errors start

Bench Worker + its own D1 (tuned indexes), k6 from one host. Errors in % of API requests.

Cache API on No cache (as built)

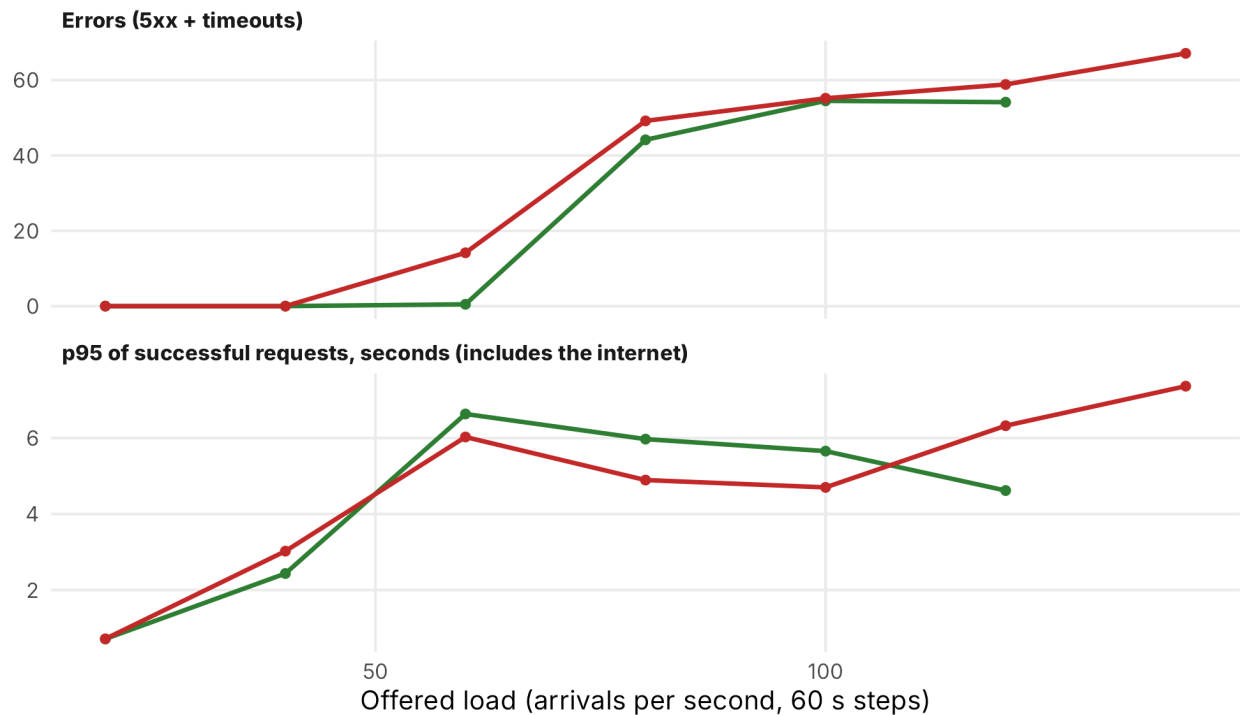


Figure 7. The edge ramp: errors and p95 of successful requests per 60 s step.

Against the local contenders, by errors alone (Table 7), the edge Worker with its cache sustained 60 arrivals per second: more than the local Workers contender (25) and less than Go (80) and Spring (90) on Postgres.

Table 7. Error-rate saturation (errors under 1%, latency ignored).

Edge, cache	Edge, no cache	Workers (local)	Go	Spring	Go + cache
60	40	25	80	90	200

H1 is partly supported. Edge caching does make the Workers implementation better under load: lower latency at every healthy step and a later error knee. It does not make it better than Go or Spring on this workload. The data tier caps it (D1 at the edge, the single D1 thread locally), the workload is 90% authenticated and personal, and the visibility projection runs on every request whatever the cache returns. And as built, the production gateway marks every feed response `Cache-Control: private, no-store`, so none of the cache arm's gains exist in production today.

4.8 Diurnal and soak (Go, no cache)

Through 24 compressed hours (peak 72 arrivals per second at 14:00 to 15:00 UTC, 0.9 times saturation) Go returned no errors, and the worst hourly p99 was 66.7 ms. By region the p99 ranged from 49.6 ms (France) to 56.5 ms (United Kingdom), a spread within the noise of a single run; India carried 34.8% of requests, the United States 14.4% and Brazil 11.8%. In the 20-minute soak p99 stayed between 50.6 and 58.1 ms per 5-minute window with no errors, and gateway memory went from 18.9 to 19.6 MB (gw1) and 17.9 to 19.3 MB (gw2): no sign of a leak.

4.9 H0

H0 cannot be rejected. Each configuration ran once, so the spread between repeated runs, which is what H0 compares against, was never measured. Request-level bootstrap intervals exist for the baselines (p99_ratio_bootstrap.csv; for example Go over Spring, p99 ratio 0.873 [0.844, 0.910]), but they ignore run-to-run variance, which on a shared host is likely the larger term, and the baselines ran at different offered loads. We report them as descriptions of one run each and call no difference significant.

5. Discussion

RQ3: what came from the runtime and what from the system. Throughput came from the system. All three gateways ran below 40% of their CPU caps when the objective broke; Postgres, D1's single thread and, in the cache arm, the object store set the ceilings. The runtime decided three other things. *Cost*: Go served about 30% more requests per gateway core than Spring and four times as many as workerd, in a twentieth of Spring's memory. *Recovery*: a JVM restart cost 8 s of cold start and 17 to 18 s to return to steady state, against well under a second for Go; a Spring fleet needs warm-up traffic or slow-start in the balancer for restarts and deploys to be invisible. *Failure shape*: under database latency the contenders fail differently at their own operating points (Section 4.6), which follows from pool sizing and offered load more than from language.

Caching moves the bottleneck, and leaves a tail. The cache more than doubled Go's ceiling and then exposed the next component, a 1-core object store, which failed within minutes at 70% of the new ceiling. Per-replica caches also make a replica restart expensive in a way a stateless replica is not: the replacement starts cold. The edge's per-data-centre cache is shared by every isolate and would not have this property, which is one of the structural advantages H1 had in mind.

Fail fast worked. On a database outage Go answered in 2 ms with a 503 instead of holding connections, readiness took the replicas out, and the system was back within a second of the link returning. That is the contract's behaviour, and the data shows it.

Indexes before runtimes. The largest effect in the study is not a runtime at all. Two indexes, no code change, multiplied Go's sustainable rate by 20 and took Spring and Workers from never meeting the objective to 90 and 15. A runtime comparison on the production schema would have measured the planner.

6. Threats to validity

6.1 The measurement path saturated first

The fault proxy sits between every gateway and its database. At 0.5 CPU and 128 MB it ran at 100% CPU in every early Go and Spring run, ran out of memory in the cache arm, and when it died the gateways logged hostname resolution errors. Those runs all "saturated near 80 arrivals per second, Postgres-bound", which was false: the proxy was the ceiling. It was found only by reading the proxy's own CPU in the cgroup samples. 14 runs with summaries (22 directories) were moved to results/_invalid_toxiproxy_cap/ and rerun at 2 CPU and 512 MB, where the proxy peaked at 23% at 150 arrivals per second and Postgres really saturated. That Go and Spring again saturated near 80 to 90 after the fix is a coincidence of this testbed, not a confirmation of the invalid runs. The lesson generalises: check the measurement path's utilisation before believing any ceiling. Section 11 of RESULTS.md lists the load generator's, proxy's, load balancer's and object store's p95 CPU for every analysed run. Two components still ran hot: HAProxy reached its 0.5 CPU cap at peaks in the Go cache arm, and MinIO saturated in it (Section 4.5).

6.2 The load generator

k6 ran out of memory above about 700 virtual users at 2 GB, about 2.87 MB per virtual user; the harness derived its VU cap from that figure. Its memory still reached 100% of its cap in the high steps of every Go and Spring ramp, well past their saturation points, so the far end of the ramps (Figure 2's flat tails) is load-generator-limited and not used for any result. An early k6 bug tainted six runs (`results/_tainted_k6bug/`), and a baseline default of a fixed 100 arrivals per second, instead of 70% of saturation, invalidated two (`_invalid_baseline_rate100/`). All were rerun.

6.3 Disk exhaustion and rebuilt images

Upload prefixes in the object store grew to 57 GB and filled the disk at about 04:15 UTC on 2026-10-02; Postgres then failed checkpoints. 52 run directories from after that time (19 with summaries, including every pass 2 and pass 3 run) were moved to `results/_failed_enospc/` and are excluded. On 2026-10-03 an unrelated session pruned every unused Docker image on the host; the gateway images were rebuilt from unchanged source, and one analysed run used the rebuilt Spring image (Section 3.10, item 9). Identical source does not guarantee identical binaries (dependency resolution, base image updates).

6.4 Single runs

Every number is one run. Run-to-run variance is unmeasured, H0 cannot be tested, and small differences (Go against Spring at steady state, RTO values within a few seconds, the region spread) should be read as observations, not effects.

6.5 Workers locally is not Workers on Cloudflare

Local workerd has no global distribution, no isolate scheduling across machines, and a local D1 whose single SQLite thread lives in one capped process. Each D1 statement crosses an HTTP hop to that store host, as in production, but the hop is local. The edge arm restores the platform but adds the public internet to every latency, and its D1 is a managed service whose limits we did not observe directly. The edge errors are 500s from the Worker; we believe D1 overload caused them, but did not confirm it.

6.6 D1 against Postgres

Workers ran on D1 (SQLite, one writer that is also the reader) and the ports on Postgres with a pooled, multi-process engine. Each is its production engine, but the comparison is of platforms, not languages, and Section 4.3 shows the storage engine dominates throughput.

6.7 The workload model

The diurnal curve weights regions by Instagram's advertising audience (DataReportal, January 2025), which counts reachable ad accounts, not active users, and shapes the hour by Sprout Social's engagement curve for brand posts, which measures engagement, not sessions. Both are proxies for request load. The request mix and the 90% signed-in share are the contract's, not measured from production traffic.

6.8 Co-tenancy and the host

The host ran unrelated production services throughout. Separate cgroup caps limit the interference but do not remove it, the load generator shares the kernel, and the network is loopback. The memory guard (abort below 1.5 GB available) never fired in an analysed run.

6.9 Port fidelity and caps

The ports are new code. Agreement and mutation tests bound how far the logic can drift; they cannot prove the ports are equally optimised. Both were written idiomatically and neither was tuned after the results were seen. The gateways never saturated, so the runtime-bound regime, where a language difference in throughput would show, was not reached; the profile designed to reach it was not run.

7. Conclusion

On one host, under hard caps, with one contract and one dataset, the three runtimes were held to nearly the same throughput by the data tier: Go 80 and Spring 90 arrivals per second on Postgres, Workers 15 on local D1. The runtime showed in cost (Go 138 requests per gateway core-second, Spring 106, Workers 35), in memory (28 MB, 594 MB, 1.37 GB for two replicas), and in recovery (Go under a second, Spring 17 to 18 s after a restart). Edge caching improved the Workers implementation on Cloudflare’s network, 37% lower median latency and a later error knee, but D1 kept it below the Postgres contenders, so H1 is partly supported. Most faults broke the latency objective while the fault lasted; errors were contained everywhere except on database faults, where failing fast is the intended behaviour. The production feed full-scans its table, and two indexes did more for throughput than any runtime. And the fault proxy, a component of the measurement itself, set the ceiling of the first set of runs until its own CPU was checked: the first thing to measure in a benchmark is the benchmark.

References

1. A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal. “Chaos Engineering.” *IEEE Software* 33(3), 2016.
2. Principles of Chaos Engineering. principlesofchaos.org.
3. G. Tene. “How NOT to Measure Latency.” (Coordinated omission.) Strange Loop, 2015.
4. B. Beyer, C. Jones, J. Petoff, N. R. Murphy (eds.). *Site Reliability Engineering*. O’Reilly, 2016. Chapters on SLOs and error budgets.
5. Grafana k6 documentation: open and closed workload models.
6. JEP 444: Virtual Threads (Java 21).
7. DataReportal. “Instagram users, stats, data and trends.” 2025-03-12.
8. Sprout Social. “Best times to post on Instagram.” 2026.
9. J. D. C. Little. “A Proof for the Queuing Formula: $L = \lambda W$.” *Operations Research* 9(3), 1961.

Appendix A. Reproducibility

Code. /srv/app/FeedBench in the /srv/app repository: contract/ (the contract, schema, tuned indexes and index study), go-gateway/, MavenFeedGateway/, workers-gateway/ (the production TypeScript snapshot, source commit 7f40e00, with PRODUCTION.diff), harness/ (runner, k6 script, chaos, monitor, summariser, edge scripts), results/ (raw data), analysis/R/ (this paper’s numbers), paper/.

Versions. Go 1.24 (net/http, pgx/v5); Java 21 with Spring Boot 3.3.5 (Tomcat, virtual threads, HikariCP, Caffeine); workerd 1.20260921.1 through Miniflare 5.20260921.0-alpha, bundled with wrangler 4.136.1; Postgres 16; HAProxy 2.9; Toxiproxy 2.9.0; Redis 7; MinIO (bitnamilegacy/minio); k6 v2.3.0; stress-ng (alexeiled/stress-ng); R 4.4 (rocker/tidyverse:4.4, data.table, ggplot2, ragg).

Images used by the analysed runs (from each run's `run.json`): Go gateway sha256:c9f7b848f52c... (every Go run); Spring gateway sha256:9b57f616e341... (every Spring run except dlatency) and sha256:411a3a935b05... (dlatency, rebuilt); Workers replicas on node:22-slim sha256:88f8ba583a88... with the gateway bundle of SHA-256 67d76ab860c5... (workers-build.json). The full list is `analysis/out/tables/runs.csv`.

Caps (analysed runs): gateways 2 x (1.5 CPU, 1 GB); HAProxy 0.5 CPU, 128 MB; Postgres 2 CPU, 1.5 GB; MinIO 1 CPU, 512 MB; Redis 0.5 CPU, 256 MB; Workers store host 2 CPU, 1.5 GB; Toxiproxy 2 CPU, 512 MB (Go, Spring) or 0.5 CPU, 128 MB (Workers); k6 2 CPU, 2 GB; stress-ng 2 CPU pinned. Gateway environment: pool 32, request timeout 5 s, feed section budget 400 ms, cache 256 MiB.

Commands.

```
cd /srv/app/FeedBench/harness
./run.sh media && ./run.sh workers seed && ./run.sh go seed      # once
./run.sh go ramp --arm nocache --schema tuned --run-id p1      # RAMP_START=20 RAMP_STEP=10
./run.sh go baseline --arm nocache --schema tuned --run-id p1 --duration 300
./run.sh go kill --arm nocache --schema tuned --run-id p1      # term cpu dlatency about objlatency
PEAK_FRACTION=0.9 ./run.sh go diurnal --arm nocache --schema tuned --run-id p1
./run.sh go soak --arm nocache --schema tuned --run-id p1 --duration 1200
edge/run-all-edge.sh                                         # edge arm (deploys the bench Worker)

cd /srv/app/FeedBench
docker run --rm --cpus 1 --memory 3g -v /srv/app/FeedBench:/fb -w /fb rocker/tidyverse:4.4 \
  Rscript analysis/R/run_all.R
```

The full queues are `harness/queue.sh` and `harness/queue-rerun.sh`, with their logs in `results/_queue.log` and `results/_edge_queue.log`.

Appendix B. How the key metrics are computed (R)

The listings below are copied from `analysis/R/metrics.R`.

```
# A request failed if the server errored (5xx) or never answered (status 0).
is_failure <- function(status) status >= 500 | status == 0

lat_summary <- function(x) {
  n <- nrow(x)
  if (!n) return(NULL)
  fail <- is_failure(x$status)
  ok <- !fail
  list(n = n, p50 = q7(x$duration_ms, .50), p90 = q7(x$duration_ms, .90), p95 = q7(x$duration_ms, .95),
       p99 = q7(x$duration_ms, .99), p999 = q7(x$duration_ms, .999), mean = mean(x$duration_ms),
       p50_ok = q7(x$duration_ms[ok], .50), p95_ok = q7(x$duration_ms[ok], .95), p99_ok =
         q7(x$duration_ms[ok], .99),
       err_rate = mean(fail), errors = sum(fail),
       rate_429 = mean(x$status == 429), rate_4xx = mean(x$status >= 400 & x$status < 500),
       apdex = (sum(ok & x$duration_ms <= APDEX_T_MS) + sum(ok & x$duration_ms > APDEX_T_MS & x$duration_ms <=
         4 * APDEX_T_MS) / 2) / n,
       budget_burn = mean(fail) / ERROR_BUDGET)
}

# Saturation under several definitions, so the choice is visible rather than buried.
# step      (primary) the highest step whose step-aggregate p99 < 500 ms, error rate < 1%
#           and no dropped arrivals: the SLO evaluated over the window, as an SLO is.
#           This is also the harness's rule (summarize.py), so the two must agree.
# knee      the last step before the first step that fails the primary rule (a later
#           step can pass again by luck; the knee is the conservative reading)
# no_drop    the primary rule without the dropped-arrivals condition
# per_second every per-second p99 inside the step < 500 ms and step errors < 1% (the
#           original R rule; at tens of requests per second a one-second p99 is the
```

```

#           single slowest request, so this rule is strict by construction)
#   error_only step error rate < 1%, latency ignored: the only rule that can compare the
#           edge (whose every request crosses the public internet) with local runs
saturation_defs <- function(steps) {
  if (!NROW(steps)) return(data.table(step = NA_real_, knee = NA_real_, no_drop = NA_real_, per_second =
    NA_real_, error_only = NA_real_,
    error_only_knee = NA_real_, first_error_step = NA_real_, steps_run = 0L,
    max_step = NA_real_))
  steps <- steps[order(step)]
  pass_step <- steps$p99 < SLO_P99_MS & steps$err_rate < SLO_ERR & (is.na(steps$dropped) | steps$dropped == 0)
  pass_nodrop <- steps$p99 < SLO_P99_MS & steps$err_rate < SLO_ERR
  pass_sec <- steps$max_sec_p99 < SLO_P99_MS & steps$err_rate < SLO_ERR
  pass_err <- steps$err_rate < SLO_ERR
  hi <- \(\p) if (any(p)) max(steps$step[p]) else NA_real_
  knee <- \(\p) { f <- which(p)[1]; if (is.na(f)) max(steps$step) else if (f == 1) NA_real_ else steps$step[f
    - 1] }
  data.table(step = hi(pass_step), knee = knee(pass_step), no_drop = hi(pass_nodrop),
    per_second = hi(pass_sec), error_only = hi(pass_err), error_only_knee = knee(pass_err),
    first_error_step = if (any(!pass_err)) steps$step[which(!pass_err)[1]] else NA_real_,
    steps_run = nrow(steps), max_step = max(steps$step))
}

# RT0 over a rolling 10 s window of API requests: seconds from the fault being lifted to the
# start of the first 10 s window, entirely after the lift, whose throughput is within 5% of
# control and whose p95 and p99 are within 10% of control's. Per-second percentiles are not
# used: at tens of requests per second a one-second p99 is the slowest request, and
# per-second throughput swings about 15%, so a per-second test almost never passes. NA means
# the system never recovered inside the observation window; it is reported, never imputed.
rto_of <- function(api, fe, hold = RT0_HOLD_S) {
  ctl <- api[phase == "control"]
  c_tp <- nrow(ctl) / uniqueN(ctl$t_sec)
  c95 <- q7(ctl$duration_ms, .95); c99 <- q7(ctl$duration_ms, .99)
  per <- split(api$duration_ms, api$t_sec)
  secs <- as.integer(names(per))
  rto <- rto_p99 <- NA_real_
  for (w0 in seq(ceiling(fe), max(secs) - hold + 1)) {
    w <- unlist(per[as.character(w0:(w0 + hold - 1))], use.names = FALSE)
    if (!length(w)) next
    p95 <- q7(w, .95); p99 <- q7(w, .99)
    if (is.na(rto_p99) && p99 <= 1.10 * c99) rto_p99 <- w0 - fe
    if (is.na(rto) && abs(length(w) / hold - c_tp) <= 0.05 * c_tp && p95 <= 1.10 * c95 && p99 <= 1.10 * c99)
      rto <- w0 - fe
    if (!is.na(rto) && !is.na(rto_p99)) break
  }
  list(rto_s = rto, rto_p99_only_s = rto_p99, control_rps = c_tp, control_p95 = c95, control_p99 = c99)
}

# MTDD: fault start to the first HAProxy sample (1 s sampling) that shows the faulted replica
# DOWN. For a fault with no replica target (database, object store) it is the first replica
# marked DOWN, if any. Resolution: the sampling interval, so a value is an upper bound to
# within about one second. Also: the time from the fault being lifted to the first sample
# showing the target UP again (re-admission).
mtdd_of <- function(d, m) {
  f <- file.path(d, "haproxy.csv")
  if (!file.exists(f) || is.na(m$fault_start_ms)) return(list(mtdd_s = NA_real_, readmit_s = NA_real_,
    hap_interval_ms = NA_real_))
  h <- fread(f, showProgress = FALSE, colClasses = list(character = c("server", "status")))
  h <- h[server %in% c("gw1", "gw2")][order(ts_ms)]
  tgt <- if (!is.na(m$fault_target) && nzchar(m$fault_target) && m$fault_target %in% c("gw1", "gw2"))
    m$fault_target else c("gw1", "gw2")
  down <- h[ts_ms >= m$fault_start_ms & server %in% tgt & startsWith(status, "DOWN")]
  mtdd <- if (nrow(down)) (down$ts_ms[1] - m$fault_start_ms) / 1000 else NA_real_
  readmit <- NA_real_
  if (nrow(down) && !is.na(m$fault_end_ms)) {
    srv <- down$server[1]
    up <- h[ts_ms >= m$fault_end_ms & server == srv & startsWith(status, "UP") & !grepl("/", status)]
    if (nrow(up)) readmit <- (up$ts_ms[1] - m$fault_end_ms) / 1000
  }
  ivl <- h[server == "gw1", median(diff(ts_ms))]
}

```

```

list(mttid_s = mttid, readmit_s = readmit, hap_interval_ms = ivl, down_server = if (nrow(down)) down$server[1]
     else NA_character_)
}

# Efficiency: requests per gateway core-second over the control phase, from the cgroup
# samples themselves (cores x sampling interval, summed). The harness's per-second
# gateway_cores series has gaps where the 1 Hz sampler missed a second, and summing it
# without na.rm made Spring's efficiency NA in the first analysis; integrating the samples
# over their own timestamps has no such gap.
efficiency_of <- function(st, r, m) {
  if (is.null(st)) return(NULL)
  ctl <- st[phase == "control"]
  if (!nrow(ctl)) return(NULL)
  dt <- ctl[, .(dt = median(diff(sort(unique(ts_ms))))), by = container][, median(dt)] / 1000
  t0 <- min(ctl$t); t1 <- max(ctl$t)
  gw <- ctl[role == "gateway", sum(cpu_cores, na.rm = TRUE)] * dt
  sut <- ctl[!role %in% c("k6", "toxiproxy"), sum(cpu_cores, na.rm = TRUE)] * dt
  n_api <- r[kind == "api" & phase == "control" & t_sec >= floor(t0) & t_sec <= ceiling(t1), .N]
  n_all <- r[phase == "control" & t_sec >= floor(t0) & t_sec <= ceiling(t1), .N]
  gw_mem <- ctl[role == "gateway", .(m = mean(mem_bytes, na.rm = TRUE)), by = container][, sum(m)] / 2^30
  list(control_s = t1 - t0 + dt, sample_s = dt, gateway_core_s = gw, sut_core_s = sut,
        gateway_cores_mean = gw / (t1 - t0 + dt),
        api_per_gateway_core_s = n_api / gw, all_per_gateway_core_s = n_all / gw,
        api_per_sut_core_s = n_api / sut, gateway_mem_gb = gw_mem,
        api_rps_per_gateway_gb = (n_api / (t1 - t0 + dt)) / gw_mem)
}

```

Appendix C. Excluded data

Directory	Run directories	With a summary	Why excluded
results/_tainted_k6bug/	6	4	a k6 script bug; rerun
results/_invalid_baseline_rate100/	2	2	baseline ran at a fixed 100 arrivals/s, not 70% of saturation
results/_invalid_toxiproxy_cap/	22	14	Toxiproxy at 0.5 CPU was the ceiling (Section 6.1)
results/_failed_enospc/	52	19	ran after the disk filled (Section 6.3)
results/<c>/smoke/	22		pipeline checks, not experiments
results/spring/kill/nocache_chaostest	1	1	dry run of the chaos pipeline
results/spring/dbout/nocache-tuned_p1	1	0	stopped by the owner mid-run