

Overshoot, Retries and Clocks: An Empirical Study of Redis-Based Rate Limiting Under Concurrency

Yash Suthar

Independent researcher, Mumbai, India · hello@yashsuthar.dev

ORCID 0009-0004-6775-1113 · DOI 10.5281/zenodo.23467402

October 2026

Abstract

Rate limiters that share state through Redis are everywhere, and the guidance for building them warns, qualitatively, about races and burst behaviour. We measure those hazards. We implement eight common strategies — check-then-act, increment-then-check, optimistic WATCH/MULTI transactions with two retry budgets, and Lua scripts for fixed windows, token buckets, sliding-window logs and sliding-window counters — and evaluate them in 3,853 trials across concurrency (1 to 64 workers), network delay (round-trip times of 0.04 to 6.73 ms), adversarial and random arrival patterns, and clock skew. Under concurrency, every atomic strategy admitted exactly the limit in all of its trials. Check-then-act over-admitted at every concurrency level above one, by 29% on average with 64 workers (worst trial: 37 extra requests on a limit of 100), and under network delay its error follows a simple lockstep model, $\lceil L/c \rceil - L$. Optimistic transactions avoided over-admission but, with a 10-retry budget, refused up to 42% of the capacity the limit promised, and under contention ran 12 to 30 times slower than a Lua script on the same key. Even with atomic decisions, fixed windows and token buckets configured as L per W admitted twice the limit within one window, and the sliding-window counter up to 1.66×; only the sliding-window log was exact. Finally, a token bucket fed timestamps by its callers admitted only 16% of its configured rate near the limit at ± 1 s of skew, while the same bucket on Redis's clock stayed exact. We turn the results into five rules for building limiters and release the harness, raw data and analysis.

Keywords: rate limiting, Redis, Lua, concurrency, race conditions, token bucket, sliding window, distributed systems

1 Introduction

Rate limiters sit in front of almost every public API. They protect databases and downstream services from bursts, keep one tenant from starving the others, and turn pricing tiers into enforceable quotas. When an API runs on more than one process, the limiter's state has to live somewhere every process can reach, and Redis is a common choice: it is fast, it offers atomic counters and scripting, and every major language has a client for it. Practitioner guides, Redis's own tutorial and large production systems all describe Redis-backed limiters [1, 2, 3].

Getting such a limiter right is subtle. Redis executes each command atomically, but a limiter needs to read, decide and write, and the obvious way to do that — read the counter, compare it with the limit, increment it — is a race between concurrent requests. The documentation warns about this and recommends Lua scripts [4]; optimistic transactions with WATCH are another documented option [5]. What the guidance does not provide is measurement. How many extra requests does the race let through, and how does that grow with concurrency and

with network delay? What does correctness cost under contention? And once each decision is atomic, does the algorithm actually keep the promise a limit like “100 requests per second” seems to make?

This paper answers those questions empirically. We implement eight strategies in the forms that guides and libraries use — two non-atomic, two optimistic, four as atomic Lua scripts covering fixed windows, token buckets, sliding-window logs and sliding-window counters — and measure them in 3,853 trials across four experiments. We ask five questions:

- **RQ1.** How far does a check-then-act limiter over-admit as concurrency and network delay grow, and can the error be predicted?
- **RQ2.** What do optimistic transactions cost in retries, latency and wrongly rejected requests?
- **RQ3.** What do the strategies cost in throughput and latency, with and without contention?
- **RQ4.** Which algorithms keep the promise of at most L admissions in *any* window of length W ?
- **RQ5.** How sensitive is a token bucket to clock skew when timestamps come from the callers?

Our contributions are:

- **Measurements of admission error.** Every atomic strategy admitted exactly the limit in all 2,520 of its trials. Check-then-act over-admitted at every concurrency level above one, reaching 28.7 extra admissions on a limit of 100 with 64 workers (worst trial +37).
- **A model of the race.** With network delay, concurrent workers fall into lockstep, and the over-admission follows $c \cdot \text{ceil}(L/c) - L$; the measured means agree with the model to within about one request at 8 workers and more.
- **The cost of optimism.** Optimistic transactions stayed correct only with a large retry budget; with a budget of 10 they refused up to 42% of the capacity the limit promised, and under contention their throughput fell 12 to 30 times below a Lua script on the same key.
- **The any-window promise.** Fixed windows and token buckets configured as L per W admitted twice the limit within one window; the sliding-window counter up to 1.66×; only the sliding-window log was exact.
- **Clocks.** Taking timestamps from callers broke the token bucket in both directions. Under overload it admitted up to 1.3% more than its rate; near the limit, with ± 1 s of skew between workers, the unguarded form admitted only 16% of its rate. The same bucket on Redis’s clock stayed within 0.1% in every setting.
- **An open artifact** containing the harness, all raw data and the analysis that generates every number in this paper.

Section 2 gives background on rate-limit semantics and Redis’s execution model. Section 3 describes the testbed and experiments, Section 4 the results, and Section 5 turns them into recommendations. Section 6 discusses threats to validity, Section 7 related work, and Section 8 concludes.

2 Background

2.1 The promise a limit makes

A rate limit of L requests per window W is a promise to the protected service: no client identity will be admitted more than L times in the period the limit describes. Two readings of “the period” are common, and they differ.

- **Per window.** At most L admissions in each fixed window $[kW, (k+1)W)$. This is what a fixed-window counter implements.
- **Any window.** At most L admissions in *any* interval of length W . This is what operators usually mean when they size a downstream system (“the database can take 100 writes a second from one tenant”), and it is the reading we use to judge burst behaviour.

We measure two kinds of error against that promise. **Over-admission** (overshoot) is the number of admissions above L ; it lets load through that the limit was meant to stop. **False rejection** is the number of requests refused while capacity remained; it fails clients the limit was meant to serve. A correct limiter has neither.

2.2 Redis’s execution model

Redis executes commands one at a time, so each single command is atomic: two concurrent INCRs on a key always produce two distinct results [4]. A *sequence* of commands is not atomic. Redis offers three ways to make several steps behave as one.

- **MULTI / EXEC** queues commands and runs them together, but the client cannot read a value inside the transaction and branch on it, so it cannot express “increment only if below the limit” on its own [5].
- **WATCH** adds optimistic concurrency: the client watches a key, reads it, and submits a MULTI / EXEC that Redis aborts if the key changed in between. The client then retries [5].
- **Lua scripts** (EVAL / EVALSHA) run to completion before Redis serves any other command, so a script can read, decide and write as one step in a single round trip [6]. Since Redis 5 a script is replicated by its effects rather than re-executed on replicas, which makes non-deterministic calls such as TIME safe to use before writes; Redis 7 removed verbatim script replication entirely [6].

2.3 Algorithms

We study the four algorithm families that practitioner guides and Redis’s own tutorial describe [1, 2].

- **Fixed window.** A counter per window, incremented on each request and compared with L . Simple and constant in memory, but a client can spend its whole allowance at the end of one window and again at the start of the next.
- **Token bucket.** A bucket of capacity B refills at rate r ; each request spends a token. Configured as $B = L$ and $r = L/W$, it admits L per W on average while deliberately allowing bursts of up to B . The token bucket is equivalent to the leaky-bucket meter described by Turner [7, 8].
- **Sliding-window log.** A sorted set of admission timestamps; a request is admitted if fewer than L timestamps fall in the last W . Exact under the any-window reading, at a memory cost proportional to L per key.

- **Sliding-window counter.** Two fixed-window counters, combined by weighting the previous window's count by the fraction of it that still overlaps the sliding window [3]. Constant memory; exact only if requests in the previous window were spread uniformly.

3 Methodology

3.1 Testbed

All experiments ran on one virtual machine with 4 vCPUs (Intel Xeon, 2.1 GHz) and 15 GB of memory, running Linux 6.18. The store was a single Redis 7.0.15 instance with persistence disabled (`save ""`, `appendonly no`), dedicated to the experiments and flushed between configurations. Clients were written in Go 1.24.7 with the go-redis v9.7.3 client. Each configuration opens a pool with one connection per worker, plus two, and pings every connection before measuring, so no trial pays for connection setup. Client retries on network errors are disabled, because a silently retried INCR would count a request twice.

Network delay. To study how round-trip time widens race windows, the client can reach Redis through a user-space TCP proxy that holds every chunk of bytes for a fixed time in each direction while preserving order. Sleeps in user space are coarse, so the delay the clients experienced is larger than the delay requested. We therefore report the **measured** median round-trip time of a PING through the same path, taken before each configuration, rather than the nominal setting. The three settings measured 0.04 ms (no added delay), 2.45 ms and 6.73 ms.

3.2 Strategies

Table 1 lists the eight implementations. All are short and follow the forms common in practitioner guides; the Lua scripts read the server clock with `TIME` in microseconds. The source of every strategy is in the artifact.

Table 1. Strategies compared. Round trips are for the common path, without retries.

Strategy	What it does	Round trips	Atomic decision
get-then-incr	GET; if below L , INCR (and PEXPIRE on the first)	2	no
incr-then-check	INCR; admit if the result $\leq L$; PEXPIRE on the first	1	count only
WATCH ($r = 10$)	WATCH, GET, then MULTI/INCR/PEXPIRE/EXEC; retry up to 10 times on conflict	3	yes, optimistic
WATCH ($r = 1000$)	as above, retry budget of go-redis’s own example [9]	3	yes, optimistic
lua-fixed-window	one script: clock-aligned window key, INCR, PEXPIRE	1	yes
lua-token-bucket	one script: refill from elapsed time, spend one token	1	yes
lua-sliding-log	one script: trim, count and add in a sorted set	1	yes
lua-sliding-counter	one script: weighted previous window plus current window	1	yes

When the WATCH strategy exhausts its retry budget it rejects the request; such rejections are counted as “gave up” and, when capacity remained, as false rejections.

3.3 Experiments

E1 — admission error under concurrency. A limit of $L = 100$ per day, so that no window rolls over during a trial, receives exactly 300 checks from c concurrent workers, $c \in \{1, 2, 4, 8, 16, 32, 64\}$, released together. A correct limiter admits exactly 100. We repeat each cell 20 times for each of the three network settings, shuffling the order of strategies within each trial, and record admissions, retries, give-ups and the latency of every decision.

E2 — cost. For 1.5 s per trial, $c \in \{1, 8, 32\}$ workers check as fast as they can in three scenarios: one shared key that never reaches its limit (every check admitted, maximum contention on writes), one shared key whose limit was reached at once (almost every check rejected), and one key per worker (no contention). We record throughput, latency percentiles and retries per decision, with no added delay and with the first added-delay setting.

E3 — the any-window promise. With $L = 100$ per $W = 1$ s and 8 workers, we drive three arrival patterns and measure the most admissions inside any interval of length W : (i) *boundary bursts* — one probe request, then 200 checks 30 ms before the next window boundary and 200 more 5 ms after it (for clock-aligned windows the boundary is computed from the Redis clock; for windows and buckets that start with the first request, from the probe); (ii) *burst then sustain* — 200 checks at once, then $4 \times L/W$ for 1.5 windows; (iii) *Poisson* arrivals at $1.5 \times L/W$ for 10 s. Admission times are taken at the client when each response arrives; at the round-trip times involved (≈ 0.05 ms) this can shift a count across a window edge by at most one, which we treat as measurement resolution.

E4 — clock skew. A token bucket with capacity 20 and refill 100/s is checked by 8 workers at a fixed pace for 5 s, under two offered loads: 40× the refill rate (4,000 checks/s, so the bucket is always empty) and 1.2× (120 checks/s, so it hovers near full). Worker clocks are offset evenly across $[-s, +s]$, $s \in \{0, 1, 10, 100, 1000\}$ ms. We compare the bucket on the Redis clock with two variants that take “now” from the caller: one that trusts it, and one that refuses to move the stored timestamp backwards (a common guard). Over a trial of length T a correct bucket admits $\min(\text{offered}, 20 + 100 \cdot T)$ requests; we report admitted divided by that figure.

3.4 Statistics and reproducibility

E1 uses 20 trials per cell, E2 two, E3 ten for the burst patterns and three for the Poisson pattern, E4 three. We report means with 95% confidence intervals from the Student t distribution, together with the worst trial where the worst case is the point. The harness, the raw per-trial data, the analysis script that produces every table and figure in this paper, and the environment record are available as an artifact; a complete run takes about an hour on the machine described above.

4 Results

4.1 RQ1: How far does check-then-act overshoot, and why?

Across 2,520 trials, every strategy whose decision is atomic admitted exactly 100 of the 300 checks: incr-then-check, all four Lua scripts, and WATCH with a 1,000-retry budget, at every concurrency level and every network setting. The non-atomic get-then-incr was exact only with a single worker. Table 2 shows the error with 64 workers, and Figure 1 shows how it grows with concurrency.

Table 2. Admissions above (+) or below (−) the limit of 100 with 64 concurrent workers. Mean $\pm$ 95% CI over 20 trials, worst trial in parentheses.

Strategy	RTT 0.04 ms	RTT 2.45 ms	RTT 6.73 ms
get-then-incr	$+28.7 \pm 2.6$ (max +37)	$+28.1 \pm 0.5$ (max +32)	$+27.4 \pm 0.2$ (max +28)
incr-then-check	0 (exact)	0 (exact)	0 (exact)
WATCH (r = 10)	0 (exact)	-10.0 ± 1.6	-42.0 ± 2.6
WATCH (r = 1000)	0 (exact)	0 (exact)	0 (exact)
lua-fixed-window	0 (exact)	0 (exact)	0 (exact)
lua-token-bucket	0 (exact)	0 (exact)	0 (exact)
lua-sliding-log	0 (exact)	0 (exact)	0 (exact)
lua-sliding-counter	0 (exact)	0 (exact)	0 (exact)

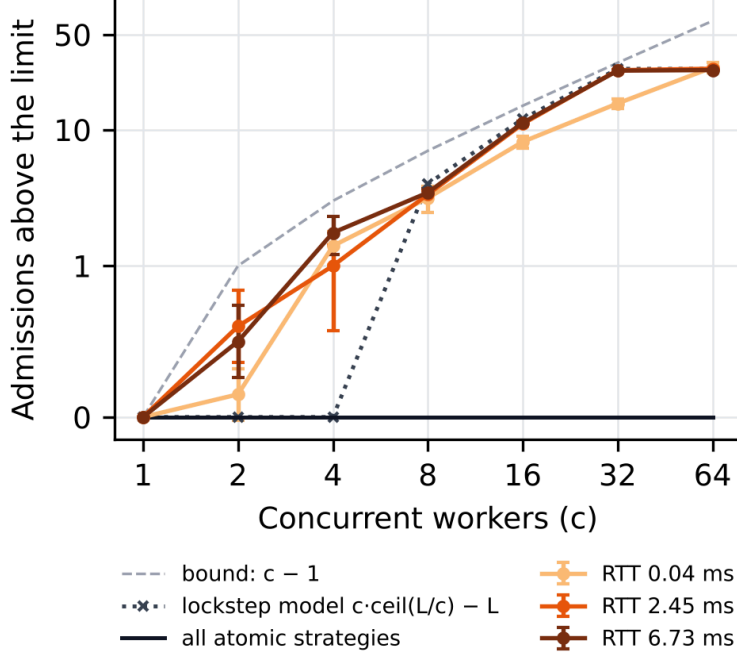


Figure 1. Admissions above the limit for get-then-incr (mean and 95% CI, log scales). Every atomic strategy stays at zero. The dotted line is the lockstep model $c \cdot \text{ceil}(L/c) - L$; the dashed line is the upper bound $c - 1$.

With no added delay the overshoot grows steadily: 3.1 extra admissions with 8 workers, 8.2 with 16, 15.6 with 32 and 28.7 ± 2.6 with 64, the last being 29% more traffic than the limit allows (worst trial +37). A single check-then-act limiter in front of a service sized for 100 requests therefore passed up to 37 more.

With network delay the picture becomes strikingly regular. At 2.45 ms and 6.73 ms the overshoot is almost the same at each concurrency level, and it follows a simple model rather than the bound. When the round-trip time is long compared with the time a worker spends between requests, the workers fall into **lockstep**: each wave of c workers reads the counter at nearly the same moment, all see the same value, and all increment. The counter then grows in steps of c , and the wave that crosses the limit admits $c \cdot \text{ceil}(L/c) - L$ requests too many. With $L = 100$ the model predicts 4 extra admissions for 8 workers, 12 for 16, and 28 for both 32 and 64. Table 3 compares this with the measurements.

Table 3. Mean overshoot of get-then-incr against the lockstep model and the bound $c - 1$ ($L = 100$).

Workers c	1	2	4	8	16	32	64
Lockstep model $c \lceil L/c \rceil - L$	0	0	0	4	12	28	28
Measured, RTT 0.04 ms	0.0	0.1	1.4	3.1	8.2	15.6	28.7
Measured, RTT 2.45 ms	0.0	0.6	1.0	3.4	11.1	27.4	28.1
Measured, RTT 6.73 ms	0.0	0.5	1.8	3.5	11.2	27.1	27.4
Bound $c - 1$	0	1	3	7	15	31	63

Two consequences follow. First, the error does not depend on how slow the network is once the workers are in lockstep, only on concurrency and on the limit; a faster network does not fix the race, it only makes the interleaving less regular, as the no-delay column shows. Second, the error depends on how L divides by c , so the same code can look correct in testing and fail

in production with a different limit or more workers: the model predicts no overshoot at all for 8 workers and a limit of 96, but 98 extra admissions (the bound $c - 1$) for 99 workers and a limit of 100.

4.2 RQ2: What does the optimistic alternative cost?

WATCH-based transactions never over-admit, because a conflicting transaction aborts and retries. The cost appears instead as retries, latency and, with a bounded retry budget, false rejections (Figure 2).

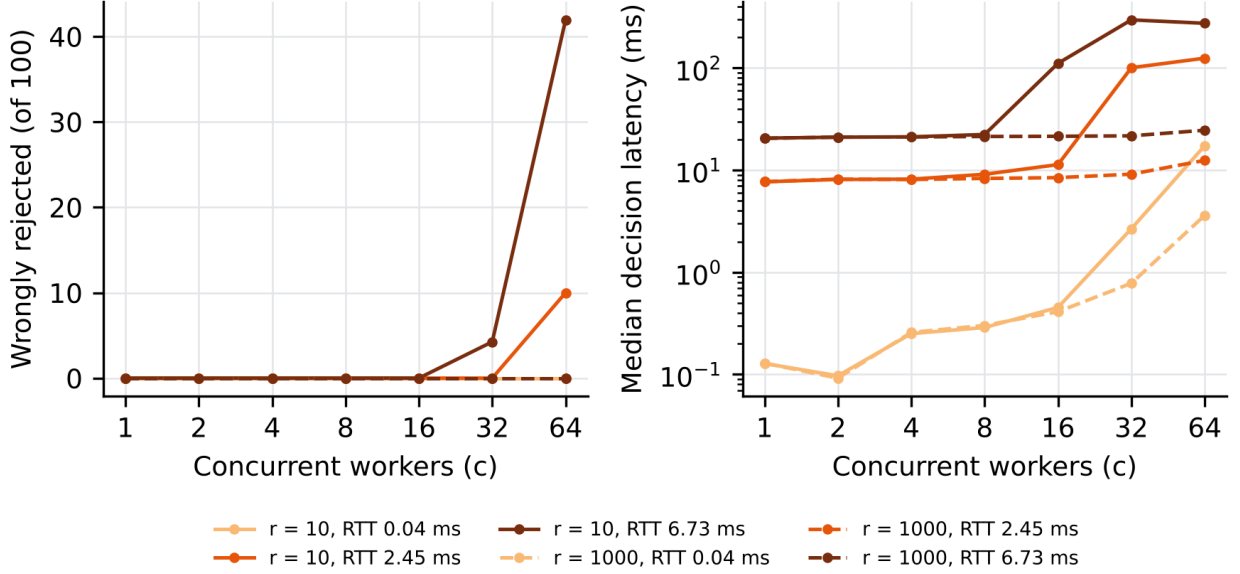


Figure 2. WATCH/MULTI against concurrency. Left: requests wrongly rejected because the retry budget ran out while capacity remained (of 100). Right: median decision latency (log scale). Solid lines: 10 retries; dashed: 1,000 retries.

With a budget of 10 retries and no added delay, WATCH still admitted exactly 100 at every concurrency level; its give-ups fell on requests beyond the limit. With delay, conflicts last longer and the budget ran out while capacity remained. With 32 workers at 6.73 ms it admitted 95.8 of 100 on average; with 64 workers it admitted 90 at 2.45 ms and 58.0 at 6.73 ms, refusing up to 42% of the capacity the limit promised. Median decision latency rose to 274.1 ms, against 6.81 ms for the Lua token bucket in the same setting. With the 1,000-retry budget used in go-redis’s example [9], WATCH admitted exactly 100 every time, but each decision needs at least three round trips, and contention adds more: median latency was 24.6 ms with 64 workers at 6.73 ms, and the 300 checks of a trial needed 5,276 retries on average. The atomic single-round-trip strategies paid one round trip regardless of contention.

4.3 RQ3: What does each strategy cost?

Table 4 and Figure 3 show throughput and latency with 32 workers. Two regimes are visible.

Table 4. Throughput (decisions per second) and median / 99th-percentile latency (ms) with 32 workers. Top: no added delay (RTT 0.03 ms). Bottom: RTT 2.44 ms. Retries per decision in parentheses where at least 0.05.

Strategy	Round trips	shared key, under limit	shared key, over limit	key per worker
get-then-incr	2	72.6k · 0.38 / 1.81	153.8k · 0.18 / 0.79	74.5k · 0.39 / 1.47
incr-then-check	1	138.6k · 0.20 / 0.86	152.1k · 0.18 / 0.74	146.6k · 0.19 / 0.80
WATCH (r = 10)	3	4.8k · 7.52 / 15.4 (6.5 retries)	48.5k · 0.57 / 2.62	33.0k · 0.86 / 3.67
WATCH (r = 1000)	3	2.6k · 8.31 / 56.0 (11.8 retries)	47.3k · 0.59 / 2.55	31.8k · 0.92 / 3.10
lua-fixed-window	1	79.9k · 0.37 / 1.12	87.0k · 0.33 / 1.13	85.9k · 0.33 / 1.67
lua-token-bucket	1	56.6k · 0.52 / 1.73	56.6k · 0.56 / 1.23	54.0k · 0.58 / 1.85
lua-sliding-log	1	56.8k · 0.52 / 2.10	79.1k · 0.37 / 1.09	57.1k · 0.52 / 1.16
lua-sliding-counter	1	59.6k · 0.51 / 1.57	74.2k · 0.40 / 1.20	61.7k · 0.51 / 0.93

Strategy	Round trips	shared key, under limit	shared key, over limit	key per worker
get-then-incr	2	5.4k · 5.83 / 8.19	10.9k · 2.92 / 5.48	5.4k · 5.86 / 8.56
incr-then-check	1	10.7k · 2.94 / 4.87	10.9k · 2.92 / 4.31	10.8k · 2.94 / 5.17
WATCH (r = 10)	3	0.3k · 127.5 / 133.6 (7.3 retries)	2.1k · 8.84 / 133.4 (0.5 retries)	2.7k · 11.8 / 14.8
WATCH (r = 1000)	3	0.1k · 83.3 / 1271.9 (15.7 retries)	1.7k · 8.84 / 412.4 (0.9 retries)	2.7k · 11.9 / 14.9
lua-fixed-window	1	10.4k · 3.06 / 4.98	10.4k · 3.06 / 5.12	10.4k · 3.07 / 5.11
lua-token-bucket	1	10.7k · 3.10 / 4.37	10.6k · 3.11 / 5.18	10.4k · 3.15 / 5.49
lua-sliding-log	1	10.4k · 3.13 / 5.06	10.5k · 3.07 / 4.36	9.7k · 3.13 / 6.29
lua-sliding-counter	1	10.4k · 3.11 / 4.88	10.5k · 3.07 / 4.64	10.4k · 3.12 / 4.94

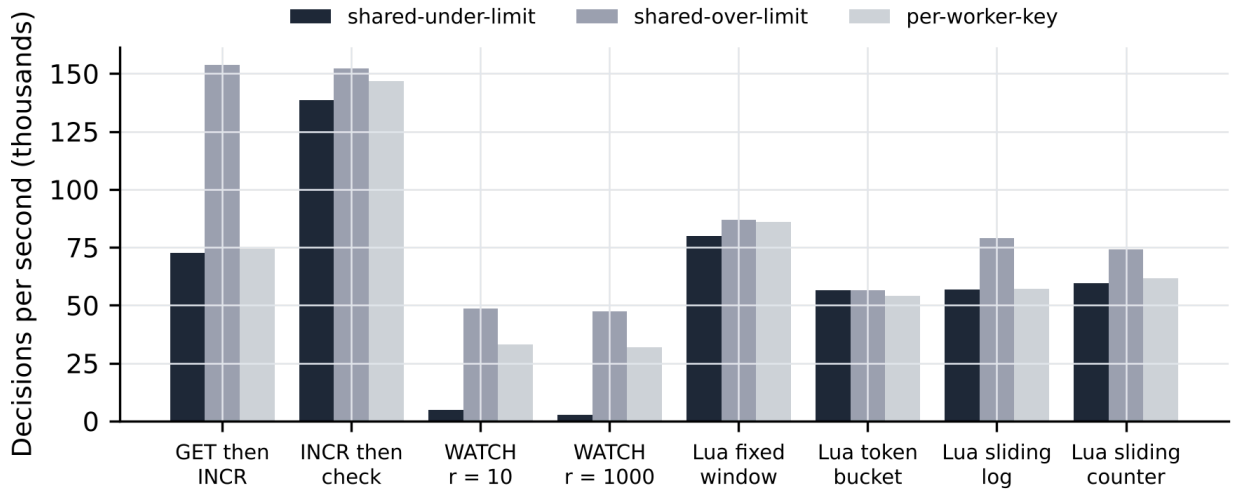


Figure 3. Decisions per second with 32 workers and no added delay, by scenario. With added delay (Table 4, bottom) every single-round-trip strategy converges on the same latency-bound throughput.

With no added delay, Redis’s CPU is the bottleneck. A single INCR is the cheapest correct decision (138.6k decisions/s on a shared key). A Lua script costs more server time per decision: the fixed-window script reached 79.9k, the token bucket 56.6k, the sliding log 56.8k and the sliding counter 59.6k, all with sub-millisecond medians. The non-atomic get-then-incr is not faster than the scripts it was presumably meant to avoid: two round trips gave it 72.6k decisions/s while it was still admitting.

With delay, round trips are the bottleneck. At 2.44 ms every single-round-trip strategy, Lua or not, delivered about the same throughput (about 10.4k to 10.7k decisions/s with 32 workers on a shared key) at a median of about one round trip, and the extra server time of Lua scripts disappeared into network time. get-then-incr halved throughput by doubling round trips (5.4k).

WATCH collapses under contention. On a key shared by 32 workers and still admitting, WATCH needed 6.5 retries per decision with a 10-retry budget and 11.8 with a 1,000-retry budget, and throughput fell to 4.8k and 2.6k decisions/s respectively, roughly 12 to 30 times fewer than the Lua scripts on the same key. With delay it fell further, to 0.3k and 0.1k, and the 99th-percentile latency of the 1,000-retry variant reached 1271.9 ms. Without contention (a key per worker) WATCH paid only its three round trips. Its cost is therefore worst exactly when a limiter matters most: many clients hitting one hot key.

4.4 RQ4: Which algorithms keep the “any window” promise?

Atomicity makes each decision correct by the algorithm’s own definition; it does not make the algorithm keep a promise of “at most L in any W ”. Table 5 and Figure 4 show the most admissions we observed in any one-second interval under a limit of 100 per second.

Table 5. Most admissions in any 1 s interval, as a multiple of the limit (100 per second). Mean $\pm$ 95% CI, worst trial in parentheses.

Strategy	boundary bursts	burst then sustain	Poisson, 1.5 $\times$ rate
incr-then-check	1.99 $\pm$ 0.00 (1.99)	1.00 $\pm$ 0.00 (1.00)	1.23 $\pm$ 0.15 (1.28)
lua-fixed-window	2.00 $\pm$ 0.00 (2.00)	1.77 $\pm$ 0.23 (2.00)	1.42 $\pm$ 0.28 (1.54)
lua-token-bucket	1.03 $\pm$ 0.00 (1.04)	1.99 $\pm$ 0.00 (1.99)	1.64 $\pm$ 0.14 (1.68)
lua-sliding-log	1.00 $\pm$ 0.00 (1.00)	1.00 $\pm$ 0.00 (1.00)	1.01 $\pm$ 0.01 (1.01)
lua-sliding-counter	1.01 $\pm$ 0.00 (1.02)	1.31 $\pm$ 0.18 (1.66)	1.26 $\pm$ 0.14 (1.31)

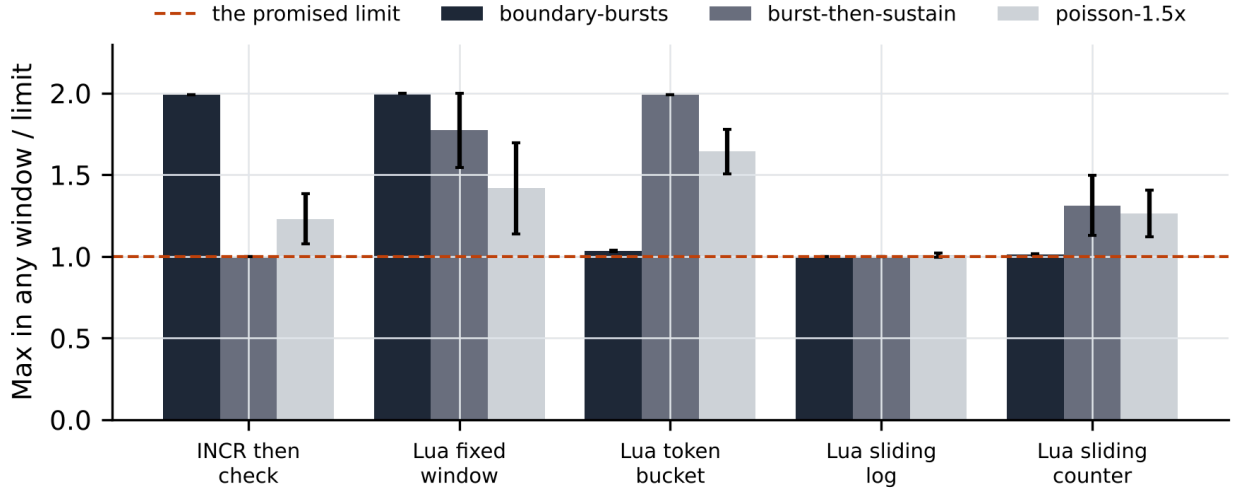


Figure 4. Most admissions in any one-second interval divided by the limit, by algorithm and arrival pattern (mean and 95% CI). The dashed line is the promise a “100 per second” limit makes.

Fixed windows admit twice the limit at a boundary. Bursts placed just before and just after a window boundary passed 200 requests within a few milliseconds through the clock-aligned fixed window ($2.00\times$ the limit), and 199 through incr-then-check, whose window starts with the first request. Even random arrivals at $1.5\times$ the limit produced intervals with $1.42\times$ the limit on average for the aligned window (worst $1.54\times$).

Token buckets admit twice the limit after a quiet spell, by design. A bucket configured as L per W (capacity L , refill L/W) absorbed the boundary bursts ($1.03\times$), but a full bucket followed by sustained demand admitted $1.99\times$ the limit in one second: the capacity plus a full window of refill. Under Poisson arrivals it reached $1.64\times$ on average. This is the correct behaviour of a token bucket, but it is not “100 per second” in the sense a downstream capacity plan usually assumes; capacity must be set below L if the any-window promise matters.

The sliding log is exact; the sliding counter is close but not exact. The sliding log never exceeded the limit by more than our one-request measurement resolution under any pattern. The sliding-window counter held the boundary bursts ($1.01\times$) but, because it assumes the previous window’s requests were spread evenly, a burst followed by sustained demand pushed it to $1.31\times$ on average and $1.66\times$ in the worst trial, and Poisson arrivals to $1.26\times$.

4.5 RQ5: What happens when callers supply the clock?

A token bucket needs “now” to compute its refill. Table 6 and Figure 5 compare a bucket that reads Redis’s `TIME` with two buckets that take the time from the calling worker, under two loads: heavy overload ($40\times$ the refill rate, so the bucket is always empty) and near the limit ($1.2\times$, so it hovers near full).

Table 6. Token bucket admissions divided by the correct number (1.000 = exact), by clock skew between the 8 workers. Mean of 3 trials; trials with the same settings agreed to within 0.006.

Load · clock	±0 ms	±1 ms	±10 ms	±100 ms	±1000 ms
40× · Redis clock (TIME)	1.000	0.999	1.000	0.999	0.999
40× · caller clock, guarded	1.000	1.000	1.003	1.011	1.013
40× · caller clock, unguarded	0.999	1.000	1.002	1.010	1.004
1.2× · Redis clock (TIME)	1.000	1.000	1.000	1.000	1.000
1.2× · caller clock, guarded	1.000	1.000	1.000	0.992	0.990
1.2× · caller clock, unguarded	1.000	1.000	0.998	0.969	0.158

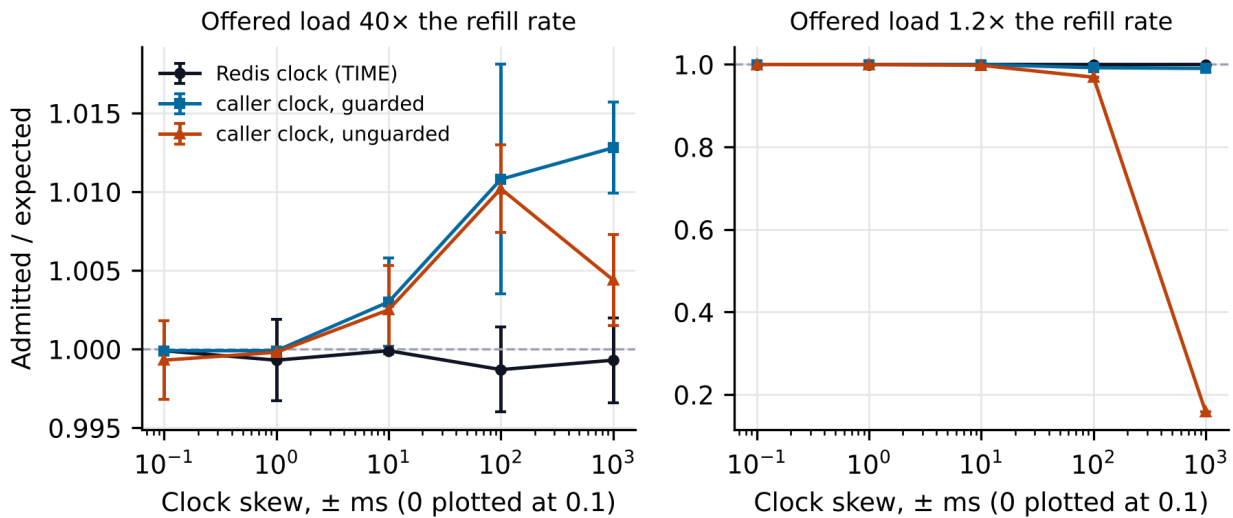


Figure 5. Token bucket admissions relative to the correct number, against the clock skew between workers. Left: overload; right: near the limit. The bucket on Redis’s clock stays at 1.00 throughout.

On Redis’s clock the bucket was exact in every setting (0.999 to 1.000). With caller clocks, the error depended on the load.

Under overload the error is small. Both caller-clock variants admitted slightly more than the rate as skew grew, up to 1.3% at ± 1 s. The unguarded bucket is nearly self-correcting here: a caller whose clock is behind subtracts tokens, a caller whose clock is ahead adds them, and while the bucket is empty the two cancel, because the sum of the elapsed times telescopes to the real interval.

Near the limit the unguarded bucket collapses. When the bucket is near full, refills are capped at the capacity but the negative “refills” computed by callers whose clocks are behind are not. Every alternation between a behind and an ahead caller then loses tokens, and the bucket drains. At ± 100 ms the unguarded bucket admitted 3.1% too little; at ± 1 s it admitted only 16% of the traffic it should have, refusing almost every request from clients that were within their limit. The guarded variant, which never moves the stored time backwards, avoided the collapse, but lost 1.0% at ± 1 s because callers behind the leading clock never earn a refill.

Clock skew of 100 ms to 1 s is not exotic: it is what a container restored from a snapshot, a host without time synchronisation, or a client-side timestamp passed through an API can produce. The fix costs nothing, since TIME is available inside the script.

5 Discussion: choosing a strategy

The results reduce to a short set of rules for anyone putting a rate limiter in front of a service with Redis.

1. **Never split the check from the update.** Check-then-act over two round trips was the only strategy that over-admitted, and it did so at every concurrency level above one. Its error is not a rounding problem: with network delay it follows $c \cdot \text{ceil}(L/c) - L$, reaching 28% above the limit with 64 workers in our runs, and up to $c - 1$ in general. It is also not faster than a correct script.
2. **Prefer a single atomic step to optimistic retries.** WATCH-based transactions are correct only with a generous retry budget, and then pay in latency and throughput under exactly the contention a limiter exists for. A Lua script, or a single INCR when the algorithm allows it, decides in one round trip with no retries.
3. **Pick the algorithm by the promise, not by the data structure.** If downstream capacity is planned as “at most L in any W ”, a fixed window or a token bucket configured as L per W can admit twice that. The sliding-window log keeps the promise exactly at a memory cost of L entries per key; the sliding-window counter keeps it approximately in constant memory, with excursions we measured up to $1.66\times$ under bursts. A token bucket is the right tool when bursts are acceptable and intended; then its capacity should be chosen as a burst budget, not set equal to L .
4. **Take time from one clock.** A script that reads Redis’s TIME makes every worker agree on “now” for free. Timestamps supplied by callers made the bucket wrong in both directions once clocks drifted apart (Section 4.5): slightly over the rate under overload, and, near the limit, far under it — at ± 1 s of skew the unguarded form refused 84% of the traffic it should have admitted. No amount of atomicity fixes an error in the inputs.
5. **Make the INCR + EXPIRE pair one step.** The incr-then-check strategy was exact in every experiment because the count is atomic, but its expiry is a second command. If a client fails between the two, the key never expires — Redis’s documentation calls this a leaked key [4] — and once its count passes the limit, that client is refused until someone deletes the key. Running both inside one script, or sending EXPIRE ... NX in the same pipeline, removes the failure mode. With network delay the script costs nothing extra (Section 4.3); without delay it costs Redis more CPU per decision than a bare INCR.

These rules say nothing about availability. When Redis is slow or unreachable, every strategy here must fall back to some policy (admit, reject, or degrade to a local limit), and that choice deserves the same care as the algorithm.

6 Threats to validity

One machine, one Redis. Clients and server shared four vCPUs, so at high concurrency the clients compete with Redis for CPU. This inflates latencies at $c = 64$ and, for the non-atomic strategy, can change how requests interleave. We do not study replicated or clustered Redis. Replication does not change the outcome of a decision on the primary, but a failover can lose recent counter updates, a separate source of over-admission that our setup cannot show.

Injected delay. The delay proxy runs in user space, and Go timers on this machine are coarse, so the requested delays (0.5 ms and 2.5 ms each way) produced the larger round-trip times we report. Because those times were measured on the same path the experiments used, the

results are valid for the measured values. The proxy may add jitter that a kernel-level emulator would not. We present the trend across the three settings, not the exact values, as the finding.

Client-side timestamps. E3 timestamps admissions when the response reaches the client, not when Redis decided. At the round-trip times involved this shifts a count across a window edge by at most one, which is why an exact limiter sometimes shows 101 admissions in a 100-per-second window.

Synthetic workloads. The arrival patterns are designed to expose worst cases (E1, E3) or to be simple to reason about (E3 Poisson, E4). Real traffic is burstier than Poisson and less adversarial than our boundary bursts. The production error rates reported for the sliding-window counter [3] and our adversarial results measure different things.

Implementations. Each strategy is a direct, short implementation of a common pattern, not a particular library. Libraries add refinements, such as EXPIRE NX pipelines or GCRA in a module [10], that move them between our categories without changing the categories' behaviour. The WATCH results depend on the retry budget; we report two budgets, one of them the value used in go-redis's own example [9].

Statistical power. E1 uses 20 trials per cell. Smaller effects, for example a mean overshoot below one request at low concurrency, have wide intervals relative to their size; we report intervals and worst trials instead of single numbers.

7 Related work

Distributed rate limiting. Raghavan et al. studied how to enforce one global limit across several sites, with the goal that congestion-responsive flows behave as if they shared a single limiter [11]. Their setting is bandwidth across a wide-area deployment; ours is request admission through one shared store, where the question is not how to split a budget across sites but whether a single counter is updated atomically.

Production designs. Cloudflare described the sliding-window counter it uses at the edge and reported a very small misclassification rate on production traffic [3]. Our burst results do not contradict that figure: they measure the approximation's behaviour under adversarial and bursty patterns, which aggregate production traffic rarely produces; under those patterns we measured up to 1.66× the limit. Stripe described layering request-rate limiters with concurrency limiters and load shedding, implemented with token buckets in Redis [2]. Redis's own documentation warns about the race in the INCR-based pattern and recommends Lua [4], and its tutorial implements every algorithm as a script [1]. To our knowledge, these sources describe the hazards qualitatively; we are not aware of published measurements of how large the errors are, how they scale with concurrency and network delay, or what the optimistic-transaction alternative costs.

Algorithms. The leaky-bucket meter goes back to Turner [7]; the Generic Cell Rate Algorithm from ATM traffic management is its single-timestamp formulation [8], and redis-cell implements it as a Redis module command [10]. A GCRA-based limiter has the same burst behaviour as the token bucket we measure, and the same need for a single trusted clock.

Clocks. That correctness should not rest on agreement between machine clocks is an old lesson [12], and it recurs in arguments about Redis-based coordination, where pauses and clock jumps break timing assumptions [13]. Our clock-skew experiment applies the lesson to a narrower case: a rate limiter whose timestamps come from its callers.

8 Conclusion

Rate limiters built on Redis fail in three distinct ways, and each can be measured. A check that is separated from its update over-admits, by an amount that grows with concurrency and that, once network delay puts the workers in lockstep, is predicted almost exactly by $c \cdot \text{ceil}(L/c) - L$. Optimistic transactions remove the over-admission but turn contention into retries, latency and, with a bounded budget, refused capacity. And atomic decisions still keep only the promise their algorithm makes: fixed windows and token buckets admitted twice the limit in a single second, the sliding-window counter up to 1.66 $\times$, and only the sliding log was exact. A single Lua script that reads the server clock and implements the algorithm whose promise matches the capacity being protected avoids all three, at the cost of one round trip.

Artifact and data

The harness (Go), the raw per-trial data for every experiment, the analysis script that produces every table, figure and number in this paper, and a record of the environment are archived with the paper on Zenodo (<https://doi.org/10.5281/zenodo.23467402>) and available at <https://www.yashsuthar.dev/papers/redis-rate-limiting-artifact.zip>. Rerunning `run_all.sh` followed by `analyze.py` and `paper/build.py` regenerates the results and the manuscript.

Use of AI tools

An AI assistant was used to help draft the experiment harness and the manuscript. All experiments were run as described above; every number in the paper is generated from the recorded data by the analysis script. The author takes full responsibility for the content.

References

1. Redis Ltd. Build 5 rate limiters with Redis: fixed window, sliding window, token bucket, and leaky bucket. Redis tutorials. <https://redis.io/tutorials/howtos/ratelimiting/> (accessed 18 October 2026).
2. P. Tarjan. Scaling your API with rate limiters. Stripe Engineering Blog, 2017. <https://stripe.com/blog/rate-limiters>
3. Cloudflare. How we built rate limiting capable of scaling to millions of domains. The Cloudflare Blog, 7 June 2017. <https://blog.cloudflare.com/counting-things-a-lot-of-different-things/>
4. Redis Ltd. INCR — including the “Pattern: Rate limiter” section. Redis documentation. <https://redis.io/docs/latest/commands/incr/> (accessed 18 October 2026).
5. Redis Ltd. Transactions — “Optimistic locking using check-and-set”. Redis documentation. <https://redis.io/docs/latest/develop/using-commands/transactions/> (accessed 18 October 2026).
6. Redis Ltd. Scripting with Lua. Redis documentation. <https://redis.io/docs/latest/develop/programmability/eval-intro/> (accessed 18 October 2026).
7. J. Turner. New directions in communications (or which way to the information age?). IEEE Communications Magazine, 24(10), 1986.
8. ITU-T Recommendation I.371. Traffic control and congestion control in B-ISDN (defines the Generic Cell Rate Algorithm).
9. Redis Ltd. Pipelines and transactions (Go client). Redis documentation. <https://redis.io/docs/latest/develop/clients/go/transpipe/> (accessed 18 October 2026).
10. B. Leach. redis-cell: a Redis module that provides rate limiting as a single command, using the generic cell rate algorithm. <https://github.com/brandur/redis-cell>
11. B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In Proceedings of ACM SIGCOMM, 2007.

12. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
13. M. Kleppmann. How to do distributed locking. 8 February 2016.
<https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html>