An Empirical Evaluation of Message Delivery Reliability and Recovery Characteristics in Redis Streams and NATS JetStream

Victor Bona
Independent Researcher, Blumenau, Brazil
victor.bona@hotmail.com

September 2026

Abstract

Consumer-progress acknowledgments in Redis Streams and NATS JetStream have different persistence implications even under always-synchronize append policies. We evaluate that distinction alongside recovery timing, consumer concurrency and publication cost using single-replica deployments on a two-node Kubernetes homelab. The study compares Redis 7.4.2/8.10.2 and NATS Server 2.10.24/2.15.0, preserving an original campaign and adding three deployments with fresh broker processes and stores, matched publication durations, CPU sensitivity and storage diagnostics. All 240 active follow-up recovery episodes completed, including live, deliberately unacknowledging consumers; twelve plain Redis read controls remained pending. Receipt timing followed residual eligibility plus policy-dependent excess. Redis's integrated CLAIM path exhibited greater excess than the tested explicit reclamation loop in this quiet, one-message workload. Sixteen consumers increased current memory/periodic drain throughput by 10.43–14.49 times across deployment-specific comparisons, while cycle p99 increased and driver capacity affected rates. At sixteen publishers, current periodic/memory publication ratios averaged 0.8257 for Redis and 0.8207 for JetStream. Ten of 72 planned always-profile publication trials failed (13.9%), including four warm-up failures, compared with one of 72 periodic trials and none of 72 memory trials. Timed failures returned five-second client timeouts with unknown confirmation outcomes. A compact analytical model and selected Lean-checked statements delimit the claims; full proofs and detailed results are included in the appendices. The licensed artifact retains timings, failures, configurations and reproducible analysis. These observations characterize specified policies and acknowledgment costs; they do not establish equal durability contracts, power-loss survival or hardware-independent rankings.

Keywords: message delivery; Redis Streams; NATS JetStream; consumer failure; redelivery latency; persistence; reproducible benchmarking.

1 Introduction

A consumer can disappear after receiving a message but before acknowledging it. Recovering that message requires more than retaining its payload: the system must preserve the pending-delivery state, make the message eligible again, and supply it to an available consumer. The resulting delay depends on both broker policy and client behavior. Conversely, a fast acknowledgment can indicate acceptance without establishing that either a message or a consumer's progress has

reached stable storage. These distinctions make a single throughput number an incomplete basis for selecting a messaging system.

Redis Streams and NATS JetStream expose different recovery policies and acknowledgment contracts. Redis consumer groups retain pending entries for explicit reclamation, including the integrated `XREADGROUP CLAIM` mechanism in current releases. JetStream tracks acknowledgment deadlines and supplies expired work when pull demand exists [25, 29, 39, 42]. Their "always" append settings also differ from a guarantee that each consumer acknowledgment synchronously persists consumer progress. A comparison must identify which operation and contract it measures.

This study asks three questions:

RQ1: How does consumer failure affect message redelivery latency?

RQ2: How do both systems behave under increasing consumer concurrency?

RQ3: What are the throughput/latency trade-offs under different durability configurations?

We operationalize RQ1 as assessment of residual timeout and recovery-policy overhead, RQ2 as acknowledged backlog-drain scaling and driver-capacity sensitivity, and RQ3 as confirmed-publication cost under specified persistence settings. The contribution is the combination of explicit acknowledgment semantics, controlled recovery interventions, and reproducible observations. It is not a new general-purpose broker benchmark or a proof of implementation correctness.

The original single-deployment campaign evaluated Redis 7.4.2 and NATS Server 2.10.24 through 300 drain trials, 60 planned publication trials, 60 serial trials, 120 consumer kills, and ten controls. Its unequal storage-history carryover motivated a prospective follow-up: three newly provisioned deployments, fresh broker processes and stores for every trial, matched publication durations, phase-specific CPU observations, and synchronization traces. The follow-up also evaluates Redis 8.10.2 and NATS Server 2.15.0, and adds live, unacknowledging recovery controls. All workloads run on the same two-node Kubernetes homelab. Historical and follow-up observations remain separate datasets.

The analytical model establishes conditional identities, bounds, and counterexamples. Measurements establish the observed behavior of the specified workloads and client paths. Neither layer supplies a hardware-independent ranking or a tested stable-storage guarantee. The distinction allows each research question to receive a concrete answer without treating empirical regularities as universal theorems.

2 Background and related work

Delivery guarantees combine distinct safety and liveness claims. Retaining an unacknowledged message is a safety property; eventually assigning it to a live consumer additionally requires a recovery policy and progress assumptions. At-least-once delivery permits duplicates and does not establish exactly-once application effects. We distinguish publication confirmation, client receipt, broker processing of a consumer acknowledgment, and completion of an external effect [3, 32]. Persistence describes which state is intended to survive a stated failure class. A consumer-process failure, broker-process failure, pod deletion, and host power loss therefore cannot be treated as interchangeable tests of "durability."

2.1 Redis Streams

`XADD` appends an entry to a stream. With acknowledgment tracking enabled, `XREADGROUP` assigns an entry to a consumer and records it in the group's pending-entry list (PEL). `XACK` removes the corresponding pending record; it does not delete the stream entry [21, 19]. A plain new-message read with `>` and no CLAIM option does not transfer another consumer's pending entries. Consequently, consumer death and message reassignment are separate events.

In Redis 7.4.2, `XAUTOCLAIM` scans pending entries and transfers those whose idle age reaches the configured minimum. It exposes a continuation cursor; a general reclaimer must advance that cursor and revisit the list. A call scans at most ten times its `COUNT` bound, so an arbitrary-size PEL cannot be treated as a single timer event [20, 25]. Our recovery experiment deliberately has one pending message and `COUNT`=1. This isolates timeout and polling behavior, but does not evaluate a large pending-list scan. Redis 8.10.2 additionally supports `XREADGROUP CLAIM`, which can consider eligible pending entries within a blocking group read [29]. The follow-up compares that policy with `XAUTOCLAIM` on both releases. A plain `>` read without CLAIM still supplies no reclamation operation.

Redis persistence is configured separately from stream consumption. We disable snapshots and compare no append-only file (AOF), AOF with `appendfsync everysec`, and AOF with `appendfsync always`. The last setting can group commands into a shared synchronization operation; it need not issue one synchronization call per message. The nominal every-second mode does not prove a hard one-second power-loss window: the pinned implementations can postpone writes while a previous background synchronization is in progress, and I/O or scheduling delay remains possible [18, 23, 26, 22].

2.2 NATS JetStream

JetStream stores messages in streams and records delivery progress in consumers. We use one durable pull consumer shared by all workers, explicit per-message acknowledgment, no acknowledgment backoff, and one stream replica. Workers bind to this same consumer instead of creating independent durable consumers, which would represent separate subscriptions rather than competing workers [36].

`AckWait` measures time from delivery, not from notification of consumer death. In the historical server, an expired pending entry is queued for redelivery; an available pull request is still necessary to obtain it. The timeout helper introduces a nominal 1 ms timer delay, and scheduling can add more [39]. Our Go client uses `AckSync`, which waits for a server response to the acknowledgment. This establishes acknowledgment processing, not an atomic business transaction and not, by itself, a synchronous stable-storage barrier for consumer state [37].

The historical server accepts `sync_interval: always` for file storage, whereas its default interval is two minutes; our periodic configuration explicitly changes that interval to one second [38, 40]. Stream append synchronization and consumer-state persistence follow different paths. For a single-replica consumer, acknowledgment updates can be handed to an asynchronous state flusher before the acknowledgment response; even always mode does not turn every such response into a synchronous consumer-state write [39, 40]. Accordingly, the two systems' "always" profiles describe append policies, not interchangeable consumer-progress persistence contracts.

The current NATS 2.15.0 source retains the distinction between synchronized stream appends and asynchronously flushed consumer state [43, 42]. We retain its synchronous single-replica append path and do not enable asynchronous stream persistence. These source facts constrain

interpretation of the acknowledgment benchmark; they do not identify the magnitude or cause of a measured throughput gap.

2.3 Prior empirical and methodological work

Broker benchmarking already includes explicit comparisons of functionality, delivery settings, and performance. Dobbelaere and Esmaili compare Kafka and RabbitMQ; Sachs et al. evaluate configurable message-oriented middleware workloads with SPECjms2007, including persistence, payload size, producer/consumer counts, and resource costs [10, 31]. These studies motivate reporting contracts and workload structure beside rates, rather than transferring historical product rankings.

OpenMessaging Benchmark already provides Redis Streams and JetStream drivers [16]. At the pinned framework revision, its Redis consumer loop uses group reads without per-message `XACK`, while its JetStream path uses a push subscription and ordinary acknowledgment. Our one-message pull/confirmed-acknowledgment cycle, controlled consumer interruption, and fresh-store procedure measure different operations. Lim et al. compare the same two products alongside RabbitMQ and Kafka for KEDA autoscaling; Paul et al. use Redis Pub/Sub and Core NATS, and Coenen et al. examine financial feeds [14, 17, 8]. These are related workloads, not replications of the present recovery and persistence interventions. We make no first-comparison claim.

At-least-once delivery does not imply exactly-once external effects [32, 3]. MillWheel illustrates stronger processing guarantees through atomic checkpoints and duplicate suppression within a defined boundary; Kafka transactions can coordinate consumed offsets and output-topic writes [1, 2]. Arbitrary external destinations require additional coordination. Our two-operation counterexamples do not contradict such transactional constructions.

Jepsen's NATS 2.12.1 technical report tests stronger faults, including simulated power failures and replicated deployments [13]. It is not a peer-reviewed comparison of our configurations, and its outcomes are not transferred to them. It reinforces the need to distinguish consumer interruption from storage failure.

Benchmark variation has multiple levels: numerous messages within a run do not replace repeated executions [12]. Randomization reduces association with execution order but cannot eliminate environmental bias [15]. Finite backlogs and closed-loop publication also differ from open-loop offered load [33]. Our request latencies are not coordinated-omission-corrected production SLA estimates [34, 11].

3 System model and formal results

Let d be broker delivery, $\widehat{d} = d + u$ client receipt, f a failure or live-control reference event, $\widehat{r}$ redelivery receipt, and $\Delta > 0$ the eligibility threshold. Initial receipt offset $u \geq 0$ is unobserved. The recovery model assumes retained payload and pending state, an available survivor, no trimming or acknowledgment extension, and broker age $h = f - d \in [0, \Delta)$. If selection occurs at $s = d + \Delta + w$, with $0 \leq w \leq P$, and subsequent observation delay is $0 \leq b \leq B$, substitution gives

$$\widehat{r} - f = \Delta - h + w + b, \qquad \Delta - h \leq \widehat{r} - f \leq \Delta - h + P + B. \tag{1}$$

The bound requires actual selection and demand-delay bounds. A configured polling sleep or observed maximum does not supply them. Without finite delay bounds, choosing arbitrarily large

w or b disproves any uniform finite recovery bound. The same algebra applies when the original consumer stays alive without acknowledging.

Receipt-based timer excess has a different origin:

$$(\widehat{r} - \widehat{d}) - \Delta = w + b - u. \tag{2}$$

It can be negative if the initial delivery-to-receipt offset exceeds subsequent overhead. Consequently, a receipt interval slightly below Δ need not violate broker-side eligibility.

For $N > 0$ completed messages, $C \geq 1$ workers and a window $T > 0$, define throughput $X = N/T$. Let message i occupy a worker for successful receive-request-to-acknowledgment time R_i, with $\overline{R} = N^{-1} \sum_i R_i$. If these intervals have disjoint interiors on each worker and lie within the window, their total length is at most CT. Thus

$$X\overline{R} \leq C, \qquad U = \frac{\sum_i R_i}{CT}, \qquad X = \frac{CU}{\overline{R}}, \tag{3}$$

where the identity requires $N > 0$ and $\overline{R} > 0$. It is an individual-trial identity, not an identity between separately averaged condition summaries. Higher concurrency need not raise throughput: $U = 1$, $(C, \overline{R}) = (1, 1)$ and $(2, 3)$ give rates 1 and $2/3$. The trace analysis checks the interval premises.

Two counterexamples delimit reliability. Repeating plain Redis new-message reads forever preserves a retained entry pending for another consumer when no reclamation occurs. Retention alone therefore does not imply reassignment. Separately, an external non-idempotent effect before acknowledgment can be duplicated by a crash and redelivery; acknowledgment before the effect can leave the effect absent. Atomic transactions, durable deduplication, and idempotency add premises outside that two-operation model.

Under the following sufficient storage-contract assumptions, confirmation implies recoverability: successful synchronization stabilizes the payload and required metadata, the failure preserves that state, recovery interprets it correctly, and confirmation follows successful synchronization. Our consumer faults do not test those storage premises. Likewise, if every counted confirmation is assigned to a completed synchronization in a serialized stage, each operation takes at least $f_{\min} > 0$, and at most $k \geq 1$ confirmations share it, that stage has rate at most $k/f_{\min}$. Overlapping stages require another model. For IID Bernoulli trials only, zero failures in n trials gives one-sided upper limit $1 - \alpha^{1/n}$, which is 0.259 at $n = 10$, $\alpha = 0.05$ [7]. Millions of dependent message completions are not millions of independent reliability trials.

Full assumptions, hand proofs, and counterexamples are provided in Appendix C. Ten selected integer-time identities, capacity statements, and finite crash witnesses are checked with Lean 4.24.0 [9, 35]; this is abstract-model verification, not verification of broker code, the harness, storage survival, or statistical coverage.

4 Experimental method

4.1 Testbed and acknowledgment contracts

All broker workloads, runtime tests, tracing, and proof compilation run through `kubectl -context homelab`. Brokers run on `homelab-01`, with a Ryzen 7 5825U, 16 logical CPUs and approximately 32 GiB RAM. The driver runs on `homelab-02`, an Intel N100 with four logical CPUs and approximately 16 GiB RAM. Both use NixOS 24.05, Linux 6.6.68, and K3s v1.30.4+k3s1.

Other homelab services remain present. Placements and resource limits control part of this shared environment; neither CPU pinning nor exclusive hardware is claimed.

Disk-backed `emptyDir` volumes use Btrfs on the broker node's NVMe device, mounted with `compress=zstd:3`. They survive container restart, but not pod deletion. Mount configuration does not establish the compression ratio of broker files or the device's power-loss behavior. Cluster-internal TCP services connect the nodes, without TLS or application authentication. Namespace ingress is restricted to experiment pods, and pods mount no service-account credentials.

The original campaign pins Redis 7.4.2 and NATS Server 2.10.24 [24, 41]. The follow-up retains those releases and adds Redis 8.10.2 and NATS Server 2.15.0, verified as current stable releases on 24 September 2026 [28, 44]. The driver toolchain and libraries remain fixed: Go 1.24.4, go-redis 9.7.0, nats.go 1.39.1 and x/sys 0.29.0. The old typed Redis decoder cannot parse CLAIM's extended reply; that arm uses a raw command with explicit four-field decoding. Image digests, module sums, runtime versions, deployed harness hashes, plans, and effective configurations accompany every follow-up deployment.

Profile	Redis	JetStream
Memory (M)	Snapshots off; AOF off	Memory stream; one replica
Periodic (P)	AOF; `appendfsync everysec`	File; `sync_interval: 1s`
Always (S)	AOF; `appendfsync always`	File; `sync_interval: always`

Table 1: Append-policy configurations, applied to both versions. Redis snapshots and automatic AOF rewriting are disabled. Streams retain acknowledged messages without configured expiration; no replication comparison is made.

A consumer acknowledgment has different persistence implications from a publication confirmation. Redis records group mutations in the AOF; the inspected JetStream releases can respond to a consumer acknowledgment before asynchronous consumer-state flushing. Furthermore, NATS 2.10.24's inspected append path ignores the synchronization return value, whereas 2.15.0 checks and propagates it; Redis 7.4.2 treats an always-mode synchronization error as fatal [23, 40, 43]. We inject no storage I/O errors and do not interpret a common profile name as an equal fault-recovery contract.

4.2 Workload and metrics

Core messages contain 512 application bytes: an eight-byte trial-unique ID and `x` filler. Protocol framing and metadata are additional and differ between systems. Each publisher uses its own connection and one outstanding confirmed `XADD` or JetStream `Publish`. Each consumer uses one-message `XREADGROUP`/`XACK` or `Fetch(1)`/`AckSync`, with no artificial application work. Workers share one group or durable pull consumer. JetStream uses explicit acknowledgments, no backoff and `MaxWaiting`=512. The original pending-acknowledgment limit is 32768; the follow-up uses 131072. Both exceed the sixteen outstanding timed consumer cycles. JetStream's normal drain acknowledgment-eligibility threshold, `AckWait`, is 30 s; it is not a workload-completion deadline.

Publication latency is request invocation to confirmation. Cycle latency is successful receive-request invocation to acknowledgment completion. Drain throughput divides acknowledged count by common drain start to the last acknowledgment; trailing empty-fetch waits are excluded. Publication trials admit requests for a fixed budget and divide confirmed count by common start to the final confirmation, including the completion tail. Admission precedes payload preparation,

so an admitted network call can begin after the budget. A nearest-rank percentile selects order statistic $\lceil pN \rceil$; reported mean p99 values average trial percentiles, rather than pooling all messages. Per-trial counts, ranks and higher-order positions are retained.

All request timestamps use the driver's Linux monotonic clock; broker telemetry uses its own node clock. Setup, warm-up, trace serialization and stored-payload readback are outside timing. Recovery survivor requests use a 100 ms block/fetch wait. Follow-up drain and victim initial reads use five seconds. Client socket/response deadlines and fetch expiry are distinct from broker eligibility; their exact version-specific behavior is documented in Appendix E [30, 37].

4.3 Original campaign and retained limitations

The original campaign uses 2-CPU/1-GiB brokers and a 2-CPU/2-GiB driver with `GOMAXPROCS=2`. It contains ten randomized complete blocks of six profiles at $C \in \{1, 2, 4, 8, 16\}$, draining 16384 preloaded messages per trial; ten five-second, sixteen-publisher publication blocks; and ten fixed-count, one-publisher blocks of 1024 messages. A 256-message warm-up uses a separate stream. ID/count reconciliation is performed, but complete payload readback is not.

Recovery uses one pending message in periodic profiles, thresholds 250/1000 ms, and kills at scheduled receipt ages 10%/75%. Redis reclamation sleeps 10 or 100 ms after unsuccessful `XAUTOCLAIM COUNT 1`, restarting from `0-0`; this is appropriate for the one-entry PEL. JetStream supplies the same durable consumer through repeated pulls. Ten episodes per cell produce 120 active recoveries; ten additional Redis controls issue only new-message reads. Historical censor endpoints were not timestamped. Their nominal budgets and final pending-state checks cannot support precise exposure-time analysis.

Logical cleanup leaves Redis's unre-written AOF growing across trials, while JetStream deletes separate stream stores. Final Redis periodic/always AOF lengths are 2,590,379,564 and 907,510,749 bytes. Original RQ3 estimates therefore include unequal storage history. Planned Redis-always trial D038 failed during timed publication before buffered client timings were serialized. A later 1568-ID store snapshot cannot reconstruct confirmations or latencies. The failed outcome remains missing numerically; its accidental completed replay remains excluded. Setup/warm-up failures and the interrupted D043 attempt are separately retained. Appendix D gives the complete retrospective attempt inventory and sensitivity diagnostics. No slow completed planned observation is removed.

4.4 Prospective follow-up and storage reset

The follow-up specifies three deployments with distinct namespaces, seeds and raw directories, and separately hashed input snapshots. Within each deployment, every trial starts a new broker process in a new exclusive store directory. The controller verifies process-group termination and a closed broker port before deleting the store. A separate 256-message stream warms publication and acknowledgment paths, is deleted, and is followed by a one-second wait. This defines the starting procedure and removes cross-trial broker-file accumulation; it does not reset host caches, device allocation history or unrelated load.

Follow-up broker pods have limits of two CPUs and 4 GiB RAM. Redis's configured memory limit and JetStream's memory-store/file-store limits are each 2 GiB (the configuration literals `2gb`/`2GB`), although these product limits account for different resources. Driver limits are 2 GiB RAM and either two or four CPUs, with matching `GOMAXPROCS`. Excluded pilots validated full-duration publication, stored-payload comparison, recovery, tracing, failure recording and

cleanup. One 1-GiB pilot exhausted memory during untimed verification; the uniform increase to 4 GiB occurred before primary collection. A separate setup race was corrected by staging, atomically renaming and hash-checking the harness before launch. Pilot failures and unexecuted cells remain documented.

Each deployment contains the following preassigned cells:

- Publication: both versions of each engine, three profiles and 1/16 publishers, with a 15-second admission budget and three randomized complete blocks: 72 trials.
- Driver sensitivity: current versions, memory/periodic profiles, $C = 1/16$, and 2/4 driver CPUs, with 65536 messages and two randomized complete blocks: 32 trials.
- Recovery: historical/current Redis XAUTOCLAIM, current Redis CLAIM, and historical/current JetStream; two thresholds, two ages, killed/live-unacknowledging victims and two randomized blocks: 80 episodes, plus four plain-read Redis controls.
- Diagnostics: current always profiles, 1/16 publishers with/without tracing, plus compressible/pseudorandom payloads crossed with inherited/disabled compression at 16 publishers: eight trials per diagnostic family.

This totals 204 cells per deployment and 612 planned follow-up outcomes. Families execute in publication, drain, recovery and diagnostic order. Cells are randomized within the stated blocks; plain-read controls follow active recovery, and the sixteen diagnostic cells are shuffled together. Publication durations and publisher counts match within comparisons. All complete publication/drain trials read back every retained ID and all 512 stored bytes outside timing. The pseudorandom arm uses a fixed-seed bank of 4096 payload templates, with the first eight bytes replaced by each ID; its template pattern repeats every 2 MiB and is not an unbounded independent random byte stream.

The recovery survivor's first request invocation is recorded before the reference event. Live victims deliberately remain unacknowledging; they do not represent normally processing applications. Actual receipt age, kill brackets, pre-reference receipts, nominal deadlines, timer-branch selection, survivor/controller completion and reconciliation endpoints are retained. XAUTOCLAIM sleeps 10 ms with randomized initial phase; CLAIM uses blocking reads without that extra sleep. Plain-read controls use both Redis versions, $\Delta = 250$ ms, 10% kill age and a nominal 2Δ observation budget. All recovery arms use one pending entry, periodic persistence and no concurrent publication load.

4.5 Telemetry, diagnostics and analysis

Before/after cgroup counters bracket each measured publication or drain phase separately. Driver and broker-container CPU include all processes in their respective containers; traced broker counts also include strace. The sample durations, throttling, host CPU counters and load averages are retained. Four driver CPUs do not imply four dedicated host cores.

Diagnostic strace captures both `fsync` and `fdatasync` across broker threads. Calls are selected using broker-side wall-clock phase brackets, with boundary overlaps reported separately; driver monotonic times are never aligned to broker wall time. Summed call durations can overlap across threads and do not decompose elapsed workload time. Matched untraced trials characterize perturbation. Compression-off trials set and verify inherited `FS_NOCOMP_FL` on fresh directories and actual broker files without disabling copy-on-write or checksumming [4, 6, 5]. This tests

a policy on the same filesystem, not a second filesystem or measured extent-compression ratio. Each diagnostic condition uses a 15-second admission budget and occurs once per deployment.

Original intervals use 10000 percentile bootstrap resamples of nine or ten completed trials or paired blocks, with seed 20260923. They are nominal and conditional on the original resampling assumptions. Follow-up analysis reports each deployment's means and ranges across deployment means. Throughput contrasts divide arithmetic condition means over jointly completed paired blocks; killed-minus-live contrasts use paired mean differences. Cross-deployment point summaries weight available deployment estimates equally, and ranges span those estimates. Failed and unmatched pairs remain counted. Three recreated deployments on shared hardware do not establish population interval coverage. We neither pool the two campaigns nor interpret their differences as the causal effect of storage reset alone.

Separate offline code reconstructs IDs, counts, worker ordering, intervals, percentiles and throughput from raw traces. A second checker recalculates condition summaries and pairing arithmetic. Failed requests retain confirmed and unknown records under returned errors or caught panics; abrupt driver loss before serialization can still lose observations. Every remote invocation has an exclusive prelaunch record, and collection verifies its result and exit status. A failed reset or infrastructure error pauses collection without automatic replay. The prospective plan and development history are retained separately from the original retrospective adjudication.

5 Results

5.1 Original evidence and motivation for the follow-up

The original campaign retained 300 completed drain trials, 60 completed serial trials, 59 completed publication trials and the failed D038 outcome. All 120 active recoveries completed; ten plain-read controls remained pending. Detailed original results and attempt accounting are in Appendix D.

At $\Delta = 250$ ms, kills at 10%/75% of the receipt-based interval were followed by median delays of 226.5/63.6 ms for JetStream and 230.2/67.9 ms for Redis's 10 ms reclamation sleep. Redis's 100 ms sleep raised the late-kill median to 134.0 ms. These observations motivate matched live controls and the current CLAIM arm. Increasing consumers from one to sixteen raised original mean drain throughput by 8.59–12.94 times across profiles, with higher cycle p99. Every recorded worker interval satisfied Equation 3. The always-profile rates concern different consumer-progress persistence contracts.

Original periodic/memory publication ratios were approximately 0.8305 for Redis and 0.8197 for JetStream. Always-mode means were 3028 msg/s for nine completed Redis trials and 375 msg/s for ten JetStream trials. Their fixed-count, one-publisher means were 603 and 585 msg/s, respectively. Unequal storage history, mismatched serial duration, missing D038 timings and influential tail observations prevent a causal interpretation of these contrasts. The fresh-store experiments address selected weaknesses while preserving all historical outcomes.

5.2 Follow-up outcomes and evidence quality

All 612 planned outcomes were collected: 601 procedures completed and eleven publication trials failed (Table 3). Failures were concentrated in always profiles: ten of 72 planned attempts (13.9%), comprising four warm-up failures and six measurement-stage failures. Periodic profiles had one failure among 72 attempts (1.4%), in historical JetStream; memory profiles had none among 72.

Table 2 separates the stages. These are observed fractions of the balanced planned configuration matrix, not estimates of a universal operational failure probability.

The seven timed failures retain 14651 confirmed and 22 unknown request records. Four trials ended with Redis socket I/O timeout errors and three with NATS response timeouts. Recorded unknown-call durations ranged from 5.000 to 5.002 s; the sixteen-publisher JetStream failure retained sixteen unknown requests, so a failed trial need not represent a single failed call. A timeout records the client returning without confirmation, not the time at which the broker completed or abandoned the operation. Warm-up failures have no timed trace and cannot be counted as fifteen-second measurement windows. Appendix F.7 preserves each failure, its error class, and the distinction between client confirmations and later non-atomic inventories. No successful-workload throughput is imputed for a failed attempt.

Profile	Planned	Complete	Warm-up	Timed	Failed, %
Memory	72	72	0	0	0.0
Periodic	72	71	0	1	1.4
Always	72	62	4	6	13.9

Table 2: Publication outcomes by append profile across both versions, publisher counts and all deployments. Warm-up and Timed count failures in those stages. Failure percentages use all 72 planned attempts per profile; warm-up failures never enter the timed publication phase.

Event reconstruction checked 80263171 timed records, including those unknown outcomes. All 349 completed publication, drain and diagnostic workloads passed stored-ID and full-payload comparison. These are recorded runtime checks and offline consistency checks, not independent experimental replication or a storage-fault test. No completed slow observation was removed, and no failed primary cell was replayed into success.

Family	Planned	Complete	Failed
Publication	216	205	11
Drain	96	96	0
Recovery	252	252	0
Trace	24	24	0
Compression	24	24	0

Table 3: All planned follow-up outcomes across three deployments. A complete negative control means that the observation procedure completed with the message still pending. Failed trials remain outcomes and do not receive invented successful-workload metrics.

5.3 RQ1: live controls and current reclamation policies

All 120 killed-victim and 120 live-unacknowledging episodes redelivered the original identifier and cleared pending state. At every fixed policy, threshold, victim state and deployment, the later reference age had a shorter mean reference-to-redelivery interval. No receipt preceded the reference event; the largest kill-invocation-to-exit bracket was 2.23 ms. Live redelivery confirms that consumer death is not a necessary trigger in these configurations.

Figure 1 separates the timeout from receipt-based excess. Across deployment-specific condition means, XAUTOCLAIM excess ranged from 0.98 to 10.70 ms and JetStream excess from 1.62 to

9.80 ms. Current Redis CLAIM ranged from 20.52 to 87.83 ms under its 100 ms blocking-read policy. Its paired killed-minus-live differences ranged from −26.91 to 14.64 ms, precluding an equivalence claim from close overall averages. These are joint broker/client policy observations, not an intrinsic recovery-speed ranking or directly observed server-timer delays.

All twelve plain-read controls retained pending state without redelivery. Their recorded reference-to-survivor-completion times ranged from 0.503 to 0.699 s for a nominal 0.500 s budget. The follow-up therefore reports actual endpoints, while the original controls retain only their nominal-budget interpretation.

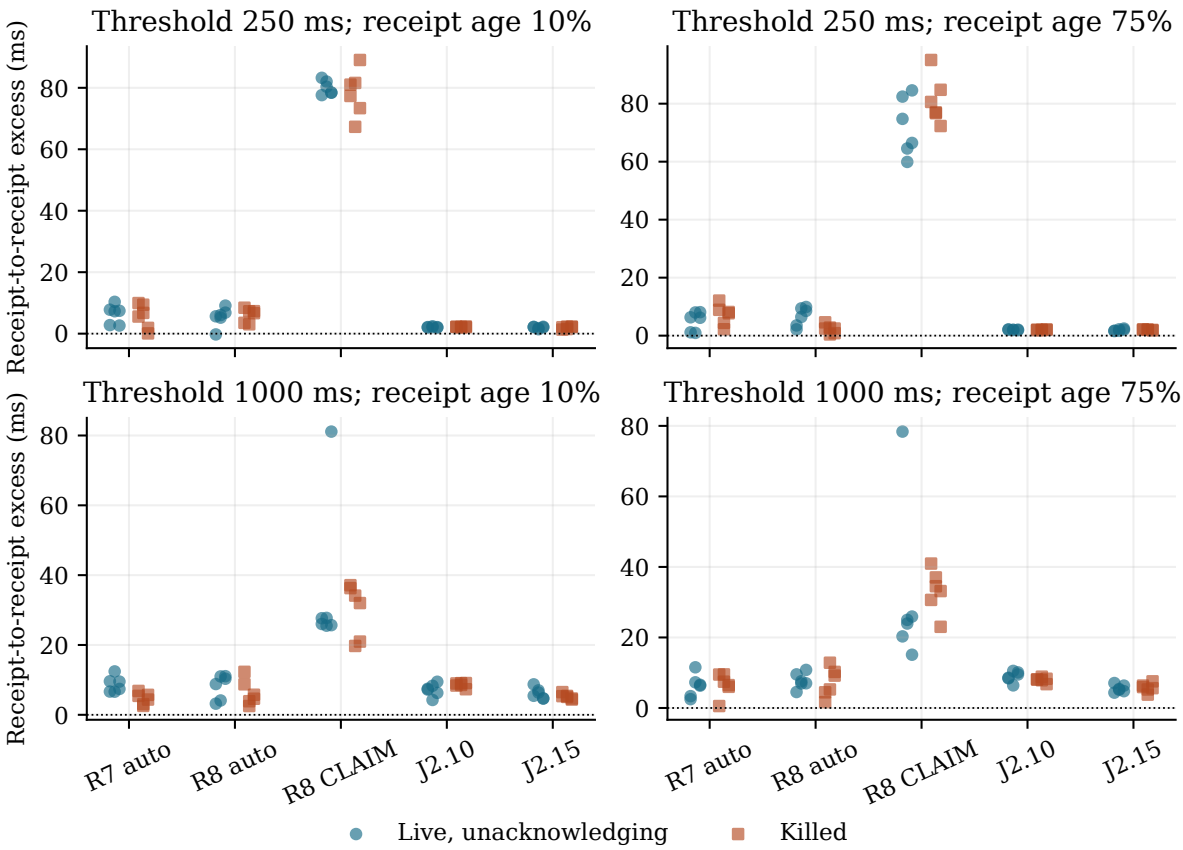

Figure 1: Every active follow-up recovery episode, showing receipt-to-receipt time minus the configured eligibility threshold. Blue circles are live, deliberately unacknowledging victims; orange squares are killed victims. Horizontal offsets separate deployments and do not encode time. R7/R8 denote Redis 7.4.2/8.10.2; auto denotes XAUTOCLAIM with a 10 ms sleep; J denotes JetStream. Zero is a client-receipt reference, not a verified broker-side lower bound.

5.4 RQ2: consumer scaling and driver capacity

All 96 follow-up drains completed. Across deployment-specific paired comparisons, sixteen consumers raised mean throughput by 10.43–14.49 times relative to one consumer in the current memory/periodic profiles. Mean trial cycle p99 increased in all eight engine/profile/driver-quota conditions. The 65536-message drains lasted 1.77–30.11 s, extending the shortest high-concurrency

observations beyond the original subsecond runs while remaining finite workloads.

At sixteen consumers, changing the driver from two to four CPUs, with matching GOMAXPROCS, produced mean paired throughput ratios of 1.031/1.026 for Redis memory/periodic and 1.107/1.102 for JetStream. Redis-periodic deployment ratios ranged from 0.993 to 1.048, so the direction was not uniform. No driver or broker cgroup throttling was recorded within the 96 bracketing drain samples. The increased rates and changed cycle tails nevertheless show that absence of throttling alone cannot establish that driver configuration is irrelevant. Table 4 reports the corresponding CPU use; these observations concern the combined client, protocol and broker path.

System	Mode	CPU	X_1	X_{16}	X_{16}/X_1	Driver	Broker
Redis	M	2	2.83	34.98	12.36 [12.02, 12.86]	1.08	0.56
Redis	M	4	2.85	36.05	12.68 [12.21, 13.52]	1.43	0.56
Redis	P	2	2.59	28.55	11.05 [10.43, 11.73]	0.98	0.64
Redis	P	4	2.55	29.31	11.48 [11.34, 11.61]	1.32	0.64
JetStream	M	2	2.28	29.54	12.98 [12.62, 13.34]	1.11	1.29
JetStream	M	4	2.28	32.67	14.32 [14.14, 14.49]	1.50	1.35
JetStream	P	2	2.26	28.70	12.72 [12.61, 12.79]	1.11	1.28
JetStream	P	4	2.28	31.62	13.89 [13.84, 14.00]	1.47	1.34

Table 4: Current-release drain results. CPU is the driver's configured quota; X_1 and X_{16} are throughput in kmsg/s. Point values equally weight campaign means. The speedup column gives the mean and range of campaign-specific paired ratios. Driver/Broker are mean CPU cores used during the bracketing samples at $C = 16$, not percentages. M/P denote memory/periodic.

5.5 RQ3: confirmed publication from fresh stores

At sixteen publishers, equal-weight deployment means for Redis 8.10.2 were 73.26, 60.49 and 4.35 kmsg/s in memory, periodic and always profiles. JetStream 2.15.0 yielded 60.62, 49.75 and 0.565 kmsg/s. Each condition completed all nine planned trials except JetStream always, with eight completions. The paired periodic/memory ratios averaged 0.8257 for Redis and 0.8207 for JetStream, with deployment ranges 0.8082–0.8397 and 0.8091–0.8296; all nine pairs were available in each comparison. These are configured confirmation costs from the specified starting procedure.

The latency trade-off remains visible. For memory, periodic and always profiles, respectively, mean trial p99 at sixteen publishers was 0.470, 0.563 and 16.98 ms for current Redis, and 0.639, 0.796 and 295.64 ms for current JetStream. A retained JetStream-always trial, main03/F0039, contained 976 confirmations and a 1968.6 ms p99, with nine higher order-statistic positions. Its tail remains part of the result. Appendix F reports every condition's completion and sample counts; these empirical tails do not establish stable production percentiles.

Long requests also occurred in completed trials. Seven completed always-profile trials had a confirmed-request maximum of at least 1000 ms, spanning both brokers and both versions. Four exceeded 2000 ms, with maxima from 2.297 to 4.183 s; the next-highest maximum was 1.969 s. Appendix F.8 lists all seven cases. These thresholds are post-observation descriptive summaries, not prospective exclusion rules or production tail guarantees. Together with the timeout outcomes, they show operationally material latency variation that is not represented by completed-run mean throughput alone.

Matched publication durations clarify the concurrency comparison that the original serial sensitivity could not isolate. For current Redis always mode, the mean paired sixteen/one-publisher ratio was 9.41, ranging from 6.06 to 13.67 across deployments, using six of nine planned pairs. Current JetStream's corresponding mean was 1.55, ranging from 0.73 to 2.46, using seven pairs. More publishers therefore did not improve every deployment's always-mode mean. Neither current release nor fresh storage removes all observed long requests or timeout outcomes. Across the eight memory/periodic version contrasts at one and sixteen publishers, equal-weight means of the deployment-specific current/historical ratios ranged from 0.9603 to 1.0232, within approximately 4% of unity. Individual deployment ratios ranged more widely, from 0.9401 to 1.1237. All contrasts used nine completed pairs except JetStream periodic at one publisher, with eight. Appendix F.3 gives the individual condition summaries. These descriptive comparisons do not establish version equivalence.

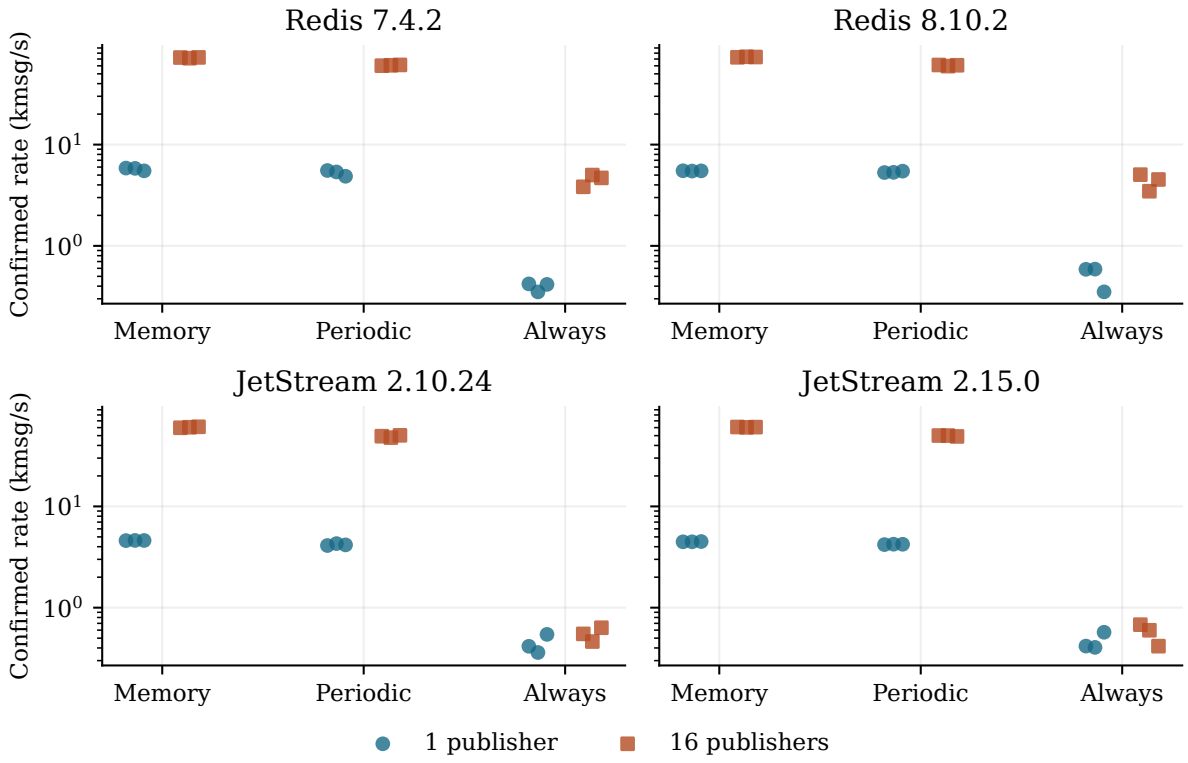

Figure 2: Fresh-store publication with matched 15-second admission budgets. Each point is one deployment's mean over completed planned trials of that condition. Horizontal offsets separate deployments; they have no quantitative meaning. The vertical scale is logarithmic. Failure counts and all per-deployment values accompany the results in Appendix F.

5.6 Synchronization and compression diagnostics

All 24 trace-family and 24 compression-family trials completed. In the twelve instrumented trials, the broker-side phase brackets contained no synchronization-call errors or boundary-overlapping calls. The synchronization-call/confirmation ratio was 1.000 for current Redis with

one publisher and approximately 0.120 with sixteen. Current JetStream's ratio was 1.000 at both publisher counts. JetStream's one-call-per-confirmation count at both publisher counts is consistent with serialized per-message synchronization in the tested single-stream append path: the pinned file store holds its write lock while storing a message, and its always-mode flush invokes synchronization before returning [43]. Redis's concurrent aggregate ratio corresponds to approximately 8.3 confirmations per synchronization call, supporting shared synchronization across publications. This identifies a source-consistent contributor to the different concurrency response; it does not assign every confirmation to an individually traced call, measure physical device flushes, or decompose the entire cross-system performance gap.

Tracing reduced throughput. Mean traced/plain ratios at one and sixteen publishers were 0.943 and 0.815 for Redis, and 0.920 and 0.934 for JetStream, respectively. Within-trace call p99 values ranged from 2.00 to 2.76 ms, and the largest traced synchronization call lasted 544.6 ms. These separate diagnostic samples did not capture the untraced primary timeout events or main03/F0039's much larger request tail, and they do not establish the cause of those events.

Disabling inherited compression produced mean off/inherited ratios of 1.036/0.995 for Redis filler/template payloads and 1.021/0.988 for JetStream. All twelve deployment-specific ratios lay between 0.935 and 1.063, with both directions represented. The complete table is in Appendix F. This sensitivity does not support attributing the large always-mode gap to compression alone; it does not establish that compression or Btrfs is irrelevant under other conditions.

System	P	Sync calls	Calls/msg	Call p99, ms	Traced/plain	n_t/n_p
Redis	1	8327 [8233, 8427]	1.000 [1.000, 1.000]	2.31 [2.23, 2.42]	0.94 [0.91, 0.97]	3/3
Redis	16	7349 [7270, 7392]	0.120 [0.120, 0.121]	2.51 [2.24, 2.76]	0.82 [0.79, 0.84]	3/3
JetStream	1	7797 [7728, 7860]	1.000 [1.000, 1.000]	2.48 [2.23, 2.66]	0.92 [0.90, 0.96]	3/3
JetStream	16	9644 [9481, 9762]	1.000 [1.000, 1.000]	2.33 [2.00, 2.60]	0.93 [0.92, 0.95]	3/3

Table 5: Current always-profile synchronization diagnostics. Entries give the mean and range of available deployment estimates; n_t/n_p counts complete traces/paired comparisons, each out of three. Call p99 is a within-trace order statistic. Traced/plain is the paired throughput ratio. Counts combine fsync and fdatasync inside the broker's phase brackets; they are not physical device-flush counts.

6 Discussion

6.1 Recovery is a joint broker/client policy

A recovery interval should be interpreted against message age, not simply against the full configured timeout. Equation 1 separates residual eligibility from selection and observation. Redis's current CLAIM option changes how an application requests reclamation; it does not make plain new-message reads reclaim another consumer's pending work. JetStream tracks expiry but still needs pull demand. Redelivery of live, unacknowledging work is therefore compatible with both policies. A shorter timeout can reduce waiting while increasing the risk of overlapping attempts when application processing exceeds it. Idempotency and external-effect coordination remain application concerns.

Redis's integrated CLAIM path checks pending-entry eligibility during event-loop processing. With little other activity, the default cron frequency supplies a nominal 100 ms wake-up interval; socket events can wake the loop earlier [29, 27]. Eligibility checks, blocking-read timeout handling,

and client reissuance can therefore contribute to receipt timing. The experiment does not instrument those internal events, so it cannot assign the observed excess to a particular scheduling component or establish an intrinsic ranking between reclamation commands.

6.2 Concurrency gains and acknowledgment cost

More consumers overlap request and acknowledgment waits, but can also increase cycle cost. Equation 3 explains why throughput and per-cycle latency can rise together. CPU telemetry and the matched driver-quota sensitivity test the available client capacity in this workload. They cannot identify an implementation-independent broker ceiling: allocation, protocol parsing, network traffic and client-library behavior remain part of the path.

The source-level acknowledgment distinction is central. JetStream's asynchronous consumer-state path prevents interpreting its fast always-profile drain acknowledgments as an equal-guarantee comparison with Redis's always-mode AOF path for group mutations. This is a semantic limit on the comparison, not a measured decomposition of its throughput gap. Publication costs are measured separately, and the current-version append error-handling difference is stated explicitly.

6.3 What the durability comparison supports

Fresh processes and stores remove the known cross-trial AOF/store accumulation in the follow-up. Matching duration and publisher count also makes its concurrency contrasts interpretable as changes within that defined procedure. The original campaign remains informative about its own accumulated-storage conditions. Its difference from the follow-up is not a reset-only intervention: duration, resource limits and other procedures also changed.

Synchronization traces describe completed software calls and their observed durations; matched untraced trials expose tracing overhead. These separate diagnostic trials cannot retrospectively trace the original incidents or the untraced primary failures. Compression-policy sensitivity tests one potential contributor on the same Btrfs device. Neither establishes that Btrfs is the dominant cause of always-mode costs, nor that each call corresponds to a distinct device flush. Such attribution would require additional controls and storage instrumentation. No publication latency, successful readback or consumer kill establishes the stable-media state after a power failure.

7 Threats to validity

Construct validity. Reliability means observed identities, confirmations, pending-state reconciliation and recovery in the specified consumer-fault model. Stored-payload readback strengthens the follow-up integrity check but does not compare every delivered payload inside the timed drain. Neither campaign tests exactly-once business effects, replicated consensus or storage-fault durability. Client receipt differs from broker delivery, and the killed-consumer reference event has a measured invocation/exit bracket.

Internal validity. Both nodes host unrelated services. CPU limits, randomization, fresh stores and phase counters do not remove shared-device load, caching, allocation history, network variation or thermal effects. The common warm-up has product-specific physical effects. Fresh broker processes prevent concurrent follow-up broker workloads, but other host activity remains. The original campaign's large unre-written AOF is a documented carryover confound. Follow-up

storage reset removes that specific accumulation; comparisons between the two campaigns cannot isolate its effect.

Measurement validity. Different client libraries and protocols remain part of the measured system. Closed-loop publication omits delays that an externally scheduled arrival process could impose. Fifteen-second windows and finite backlogs do not establish stationarity or saturation. Cgroup sample windows bracket rather than exactly equal request windows; broker-container CPU includes control and tracing processes. Syscall tracing perturbs execution. Compression flags specify policy, not the compression ratio of every extent. Requests lost through abrupt driver failure before serialization remain an unclosed recording boundary.

Statistical conclusion validity. Original bootstrap intervals depend on nine or ten completed trials and a working block-resampling model. Failure-associated missingness, serial dependence and influential tails are not repaired by more resamples. Follow-up results report three separately provisioned campaigns on the same hardware, not independent hosts, verified confidence coverage or external replication. Their ranges express observed between-campaign variation, not prediction intervals. Per-condition and paired-completion counts accompany estimates. Small empirical p99 samples remain descriptive order statistics; historical D040's p99 is the second-largest of 149 observations.

External validity. The study uses one Btrfs/NVMe environment, single replicas, 512-byte payloads, at most sixteen workers and a fixed pair of client libraries. The pseudorandom sensitivity retains a finite repeating template bank. There is no second filesystem, open-loop load, application-processing workload, large pending set, saturated recovery, geographic distribution, quorum fault or power-cut experiment. Current releases are current as of the stated review date, not indefinitely. The literature review is targeted and makes no exhaustive novelty claim.

8 Artifact availability and reproducibility

The public repository is `https://github.com/vicotrbb/redis-jetstream-reliability`. The retained versions 1.1.0 and 1.2.0 preserve their preceding manuscripts and evidence. Version 1.3.0 contains this integrated article with all proofs and detailed tables in its appendices, raw compressed timings, plans, failure/development records, environment and binary receipts, pinned source references, analysis, figures, and the selected Lean model. Original manuscript, data and narrative material use CC-BY-4.0; original software and formalization use MIT. `LICENSE.md` defines third-party exceptions. No DOI assignment or peer-reviewed publication is claimed.

Offline reanalysis checks original evidence against its frozen inventory and verifies each follow-up campaign's input, invocation, environment and raw-data records. It reconstructs IDs, counts, intervals, throughput, order statistics, paired contrasts, CPU deltas and synchronization summaries. The original and follow-up datasets remain separate. D038's missing client observations and historical censor endpoints remain unavailable. New collection requires new identities and destinations; failed attempts cannot be overwritten or automatically replayed.

Release manifests bind the exact source, complete article PDF, analysis and rendered document QA. The builder refuses version reuse and verifies a fresh archive extraction. Hashes establish consistency of supplied bytes, not independent certification of collection time or completeness. Public

access and computational reproduction are distinct from an external experimental replication or an independent artifact evaluation.

OpenAI Codex assisted with source investigation, software, analysis, formalization, drafting and internal checking. Scientific claims are grounded in cited primary sources, stated assumptions and retained measurements. AI-assisted internal review is not independent scientific peer review, and no generated benchmark value substitutes for an observation.

9 Conclusion

The comparison shows why recovery policy and acknowledgment semantics must accompany broker performance measurements. Redis's AOF path for consumer-group mutations and JetStream's asynchronously flushed consumer state do not provide interchangeable per-acknowledgment persistence contracts. Fast acknowledgments therefore cannot be interpreted as an equal-guarantee durability advantage.

For RQ1, all 240 active follow-up episodes recovered, including live, unacknowledging victims, while twelve plain Redis read controls retained pending work. Later reference events left shorter residual intervals. Receipt-based excess varied with the selected client/broker policy, including appreciably greater excess for the tested current CLAIM path. Consumer death was not required for eligibility, and no empirical hard deadline follows.

For RQ2, increasing consumers from one to sixteen raised current memory/periodic drain throughput by 10.43–14.49 times across deployment-specific comparisons, with higher cycle p99. The matched driver-capacity sensitivity changed rates, especially for JetStream. These are finite-workload scaling results for the measured client paths, not broker-capacity ceilings.

For RQ3, fresh-store, matched-duration publication retained approximately 83% and 82% of memory throughput in the current periodic profiles at sixteen publishers. Always-mode costs remained larger and more variable: ten of 72 planned attempts failed, including four during warm-up, and four completed trials contained a request exceeding two seconds. The timed failures returned client timeouts with unknown confirmation outcomes. Synchronization and compression diagnostics characterize the tested paths without establishing a dominant filesystem cause or stable-media survival.

The contribution is a reproducible comparison with explicit operation contracts, controlled recovery interventions, and preserved successful and failed outcomes. The original evidence and the three new deployments remain separately identifiable. Independent storage hosts, loaded recovery, open-loop arrivals, replication and controlled storage faults are the next empirical steps for extending these conclusions beyond the present scope.

A Complete concurrency matrix

The complete original concurrency matrix retains all thirty conditions, each with ten trials and 16384 messages. It reports trial-mean throughput with nominal bootstrap intervals and distinguishes cycle p99 from common-drain-start-to-receipt p99. Appendix F separately reports the follow-up by deployment, driver quota, concurrency and persistence profile. Machine-readable trial and condition summaries accompany both matrices in the artifact.

System	Mode	C	Throughput (kmsg/s)	Cycle p99 (ms)	Drain p99 (ms)
Redis	memory	1	2.89 [2.74, 3.01]	0.543	5649.9
Redis	memory	2	6.47 [6.06, 6.90]	0.509	2532.3
Redis	memory	4	13.21 [12.82, 13.64]	0.539	1231.2
Redis	memory	8	24.42 [23.98, 24.83]	0.566	664.4
Redis	memory	16	36.37 [35.94, 36.83]	0.867	445.8
Redis	periodic	1	2.66 [2.56, 2.76]	0.565	6116.9
Redis	periodic	2	6.34 [5.81, 6.90]	0.523	2606.0
Redis	periodic	4	12.94 [12.31, 13.62]	0.640	1261.8
Redis	periodic	8	22.28 [22.02, 22.55]	0.644	728.0
Redis	periodic	16	29.32 [27.73, 30.50]	1.334	557.6
Redis	always	1	0.29 [0.29, 0.30]	5.942	55604.9
Redis	always	2	0.34 [0.33, 0.35]	10.915	47736.5
Redis	always	4	0.67 [0.66, 0.68]	11.291	24240.7
Redis	always	8	1.31 [1.30, 1.33]	11.395	12380.0
Redis	always	16	2.51 [2.47, 2.55]	10.311	6469.7
JetStream	memory	1	2.30 [2.26, 2.34]	0.653	7052.2
JetStream	memory	2	5.46 [5.25, 5.74]	0.633	2986.6
JetStream	memory	4	11.69 [11.52, 11.90]	0.667	1388.1
JetStream	memory	8	20.32 [20.05, 20.61]	0.851	798.4
JetStream	memory	16	29.79 [29.43, 30.12]	1.310	544.3
JetStream	periodic	1	2.27 [2.24, 2.29]	0.636	7155.0
JetStream	periodic	2	5.50 [5.20, 5.84]	0.627	2970.8
JetStream	periodic	4	11.43 [11.23, 11.65]	0.664	1421.1
JetStream	periodic	8	19.55 [19.28, 19.80]	0.857	829.6
JetStream	periodic	16	28.03 [26.89, 28.97]	1.520	580.6
JetStream	always	1	2.24 [2.21, 2.27]	0.636	7245.0
JetStream	always	2	5.34 [5.15, 5.61]	0.631	3051.9
JetStream	always	4	11.50 [11.20, 11.79]	0.728	1413.7
JetStream	always	8	19.66 [19.49, 19.81]	0.850	824.5
JetStream	always	16	27.43 [27.04, 27.83]	1.541	591.9

B Serial-publisher sensitivity

The following table retains the original serial sensitivity: one publisher, one outstanding request, 1024 messages per trial and ten randomized blocks. That fixed-count experiment differs in duration from the original five-second concurrent workload. The follow-up instead compares one and sixteen publishers under the same 15-second admission budget, with completion counts and paired estimates reported separately.

System	Mode	Throughput (msg/s)	Publish p99 (ms)
Redis	memory	5,946.8 [5,385.6, 6,724.1]	0.240 [0.217, 0.257]
Redis	periodic	5,494.4 [5,076.8, 6,107.2]	0.257 [0.233, 0.278]
Redis	always	602.7 [591.3, 613.6]	2.847 [2.365, 3.340]
JetStream	memory	4,736.8 [4,479.8, 5,070.7]	0.325 [0.304, 0.351]
JetStream	periodic	4,911.5 [4,457.7, 5,425.8]	0.334 [0.304, 0.362]
JetStream	always	584.8 [564.8, 598.0]	2.756 [2.063, 3.807]

Table 7: Original fixed-count, one-publisher sensitivity.

C Formal assumptions and claim-to-evidence map

Claim	Evidence	Boundary
Residual recovery delay	Equation 1; fault/reference timestamps	Requires retained state and explicit finite selection/demand-delay bounds
No new-read reclamation	Model counterexample; plain-read controls; pinned Redis source	Specific to plain new-message reads without reclamation
Concurrency cost	Equation 3; original and follow-up drain traces	Finite workload and measured client path; no universal monotonic scaling claim
Persistence cost	Original publication/serial outcomes; fresh-store campaigns; matched durations and traces	Confirmation and configured policy, without power-loss or I/O-fault validation
Stable-storage implication	Conditional storage argument and Proposition C.6	Abstract successful-and-honored synchronization contract; no implementation verification
Application effect ambiguity	Two-order crash counterexamples and Proposition C.7	Separate non-idempotent effect and acknowledgment; not an impossibility result for transactional applications
Observed identities	ID/count checks; follow-up full stored-payload readback	Historical checks are narrower; readback is not a storage-fault test

C.1 Events, failure scope, and assumptions

An application assigns each message a unique identifier. We distinguish publication invocation p, publication confirmation q, broker delivery d, client receipt $\widehat{d}$, consumer failure f, redelivery receipt $\widehat{r}$, and acknowledgment completion a. All request and recovery event intervals use a common Linux monotonic clock on the driver node. Broker telemetry uses the broker node's clock and is never subtracted from driver timestamps. The broker's internal delivery timestamp d is not directly observed; replacing it with $\widehat{d}$ without qualification would misstate the timer origin.

The fault model is a fail-stop consumer process killed with `SIGKILL` while its message is unacknowledged. In the follow-up's matched live controls, f instead denotes the scheduled reference event while the victim remains alive without acknowledging; the same residual algebra applies without requiring a death. The broker, network, retained payload, group/consumer state, and surviving consumer remain available. There is no message expiry, trimming, competing claim, negative acknowledgment, acknowledgment extension, or backoff change during the recovery episode. The timeout threshold is $\Delta > 0$ and failure occurs at age $h = f - d$, with $0 \leq h < \Delta$. These are assumptions of the isolated formal model; they do not characterize every production execution. The controller sets age relative to client receipt, so exact broker age is unobserved. Idealized eligibility also abstracts from timer quantization and assumes no relevant broker-clock discontinuity.

For a bounded redelivery theorem, additional timing assumptions are required. Let P bound the time from eligibility until a reclaimer or expiry mechanism selects the entry, and let B bound subsequent scheduling, demand availability, transport, and client observation. Redis command execution time and scheduler delay must be included in P; its configured sleep alone is not such a bound. JetStream also requires pull demand, so B cannot silently be set to zero.

Theorem C.1 (Residual timeout and conditional recovery bound)**.** *Under the preceding assumptions, let selection be forbidden before $d+\Delta$, occur at some $s \in [d+\Delta, d+\Delta+P]$, and be observed after an additional delay $b \in [0, B]$. Then*

$$\Delta - h \;\le\; \widehat{r} - f \;\le\; \Delta - h + P + B.$$

If the corresponding upper-bound premise is removed and admissible selection or observation delays are unbounded, no finite uniform upper bound on redelivery latency follows.

Proof. By definition, $\widehat{r} = s + b$ and $f = d + h$. Hence

$$\widehat{r} - f = \Delta - h + \big(s - (d + \Delta)\big) + b.$$

The two final terms lie in $[0, P]$ and $[0, B]$, respectively, proving both inequalities. If either delay is unbounded, for any proposed finite upper bound choose an admissible delay exceeding it. The corresponding execution violates that proposed bound without violating eligibility or retention. □

Corollary C.2 (Why client-observed timer excess can be negative)**.** *Write $\widehat{d} = d + u$, where $u \ge 0$ is the initial delivery-to-receipt delay, and let $w = s - (d + \Delta)$. Then*

$$(\widehat{r} - \widehat{d}) - \Delta = w + b - u.$$

Thus a measured receipt-to-redelivery interval slightly below Δ does not alone refute server-side timeout eligibility.

Proof. Substitute $\widehat{r} = d + \Delta + w + b$ and $\widehat{d} = d + u$, and cancel d. □

Proposition C.3 (No autonomous Redis reassignment from new-message reads)**.** *In the stated Redis new-message-read-only model, there exists an infinite execution with an available broker and surviving consumer, a retained message pending for a failed consumer, and no redelivery, if the survivor performs only new-message reads and no reclamation or pending-history recovery.*

Proof. Deliver the only message to the consumer that subsequently fails, thereby placing it in that consumer's PEL history. Repeat `XREADGROUP` with `>` and no CLAIM option from the survivor. Each read considers previously undelivered entries, while the retained entry remains pending for the failed consumer. With no further publication or state-changing recovery operation, every repetition preserves this state. The infinite repetition witnesses the claim. □

The proposition is a liveness counterexample, not a defect claim. A Redis application supplies the missing recovery operation. Likewise, a JetStream pull consumer without future pull demand has no unconditional delivery-time guarantee. The experimental contrast concerns who initiates eligibility/reassignment and the timing of the configured client policy.

C.2 Concurrency and finite-window throughput

Definition C.4. For $N > 0$ completed messages, an integer $C \geq 1$ of consumers, and a drain interval of duration $T > 0$, let $X = N/T$ be acknowledged throughput. Message i occupies one consumer during a nonoverlapping-on-that-consumer interval $[r_i, a_i]$, from its successful receive-request invocation to acknowledgment completion. Write $R_i = a_i - r_i$ and $\overline{R} = N^{-1} \sum_i R_i$.

Theorem C.5 (Consumer occupancy bound)**.** *If every interval lies within the measured drain window and each of C consumers has at most one such interval active at a time, then*

$$X\overline{R} \leq C.$$

More precisely, with $U = (CT)^{-1} \sum_i R_i \in [0, 1]$, one has $X = CU/\overline{R}$ whenever $N > 0$ and $\overline{R} > 0$.

Proof. For consumer j, its intervals have disjoint interiors and are contained in a window of length T, so their lengths sum to at most T. Summing over all C consumers gives $\sum_i R_i \leq CT$. Since $X\overline{R} = (N/T)(\sum_i R_i/N) = \sum_i R_i/T$, the inequality follows. Substituting the definition of U yields the identity. □

For two individual trials with positive mean cycle times, this gives the exact explanatory decomposition:

$$\frac{X(C_2)}{X(C_1)} = \frac{C_2}{C_1} \frac{U(C_2)}{U(C_1)} \frac{\overline{R}(C_1)}{\overline{R}(C_2)}.$$

The identity does not generally survive separately averaging throughput, occupancy, and cycle time across trials. Reported ratios of condition-mean throughputs are distinct summaries. Approximately stable cycle cost and occupancy are sufficient for approximately linear scaling within this model. More generally, linear scaling requires $U(C)/\overline{R}(C)$ to remain approximately constant; increasing C alone does not prove it. For example, admissible conditions with $U = 1$, $C_1 = 1$, $\overline{R}(C_1) = 1$, $C_2 = 2$, and $\overline{R}(C_2) = 3$ give throughputs 1 and 2/3. This counterexample rules out a universal monotonicity claim under unrestricted contention. Our analysis checks the occupancy inequality directly against every retained concurrency-trial event interval.

C.3 Acknowledgment, storage, and application effects

Proposition C.6 (Conditional stable-storage implication)**.** *Assume (i) a successful synchronization makes the message and all metadata required for its recovery stable, (ii) subsequent failures preserve that stable state, (iii) recovery interprets it correctly, and (iv) confirmation occurs only after that synchronization completes successfully. Then every confirmed message is recoverable after such a failure. If confirmation is instead permitted before stabilization, confirmation alone does not imply recoverability.*

Proof. Under (iv), every confirmed message has already undergone the successful synchronization of (i). Its recovery state survives by (ii), and (iii) recovers it. For the second claim, choose an allowed execution in which confirmation precedes synchronization and a failure occurs strictly between them. Let the failure discard all unstable state. The message was confirmed but is not recoverable, providing a counterexample. □

This is a theorem about a storage contract, not a formal verification of either product. In particular, the historical NATS 2.10.24 append path invokes synchronization without checking its

returned error in that path, whereas Redis 7.4.2 treats an always-mode synchronization error as fatal [40, 23]. NATS 2.15.0 checks and propagates the corresponding append synchronization error [43]. The historical error-handling statement must not be generalized to that current release. The experiment injects no I/O faults and does not establish assumptions (i)–(iv) for every implementation execution. The source distinction prevents us from treating a mode name as an unconditional proof of identical fault handling.

Consider an interval containing $F \geq 1$ complete, nonoverlapping synchronization operations. If each occupies at least $f_{\min} > 0$ time, every counted confirmation is assigned to one completed operation, and at most an integer $b \geq 1$ of confirmations are assigned to each operation, those operations can confirm at most bF messages while occupying at least $Ff_{\min}$ time. Their service rate is therefore at most $b/f_{\min}$. This bound applies to the explicitly serialized synchronization stage; overlapping stages and the rest of the pipeline require a different model. It explains why group commit and outstanding-publication count can affect cost without asserting a fixed number of hardware synchronization calls per message.

Proposition C.7 (The application-effect/acknowledgment gap)**.** *For a non-idempotent external effect and a broker acknowledgment that are separate operations, changing only their order cannot guarantee exactly one completed effect under arbitrary fail-stop interruption and eventual redelivery of unacknowledged work.*

Proof. If the effect precedes acknowledgment, interrupt after the effect and before acknowledgment. The retained unacknowledged message may be redelivered and its effect repeated. If acknowledgment precedes the effect, interrupt after acknowledgment and before the effect; the broker has no remaining obligation to redeliver, so the effect can be absent. Either order admits a counterexample. An atomic transaction, durable application deduplication, or an idempotent effect adds an assumption not present in this model. □

C.4 What zero observed failures establishes

For $n \geq 1$ independent identically distributed Bernoulli trials with failure probability p, observing zero failures has probability $(1-p)^n$. For $0 < \alpha < 1$, setting this to α gives the one-sided exact upper confidence limit

$$p_{\text{upper}} = 1 - \alpha^{1/n}.$$

At $n = 10$ and $\alpha = 0.05$, this is approximately 0.259, not zero [7]. Our heterogeneous configurations and temporally adjacent messages are not automatically independent Bernoulli draws from one population. We therefore report exact observed counts and do not turn millions of message completions into a misleading claim of extremely small universal failure probability.

C.5 Machine-checked subset and proof boundary

Ten declarations in the artifact's Lean 4.24.0 model check integer-time residual and receipt-offset identities, a division-free consumer-capacity lemma, and finite effect/acknowledgment crash witnesses [9, 35]. The retained homelab compiler receipts identify the source, compiler, exit status, and transitive logical dependencies. No admitted proof, custom axiom, or native-evaluation shortcut is used. The capacity lemma assumes each consumer's busy-time bound; the interval argument deriving that premise remains the hand proof above. This checks selected abstract statements, not broker implementations, measurement software, storage survival, or statistical coverage.

D Original statistical method and detailed diagnostics

These details concern Redis 7.4.2 and NATS Server 2.10.24 in the original accumulated-storage campaign. They remain separate from the fresh-store follow-up in Appendix F.

D.1 Original statistical method and attempt history

The randomization seed is 20260923. The experimental unit is a trial, with pairing by block for treatment ratios. For each condition we report the arithmetic mean of its completed planned trials and a percentile bootstrap 95% confidence interval using 10000 resamples of those trial indices. Conditions have ten completed trials except Redis-always publication, which has nine completions and one failed planned trial. That failure is reported separately, not assigned an artificial throughput or latency. A paired ratio is the ratio of resampled condition means under the same resampled block indices, not the mean of arbitrary per-message ratios. Comparisons involving a failed block use only the blocks completed by both conditions, so these are conditional-on-completion estimates. The bootstrap seed is fixed. These nominal, conditional confidence intervals summarize uncertainty in the condition mean under the working resampling assumption that completed block vectors are independent, identically distributed replicates. All blocks belong to one deployment campaign; their temporal dependence and the evolving storage history are not estimated away by resampling. They are not prediction intervals for individual runs, do not have proven exact finite-sample coverage here, and do not account for every systematic error or hardware population.

These are descriptive effect estimates, not a collection of significance tests with familywise error claims. We specify no p-value threshold, do not stop when an interval becomes favorable, and do not pool individual messages as independent replications. Recovery tables report all ten values through medians and ranges rather than presenting a high percentile from ten failure episodes as a precise tail estimate. No finite trial count can prove absence of all possible losses.

Additional post-observation diagnostics assess sensitivity without changing the primary dataset: leave-one-trial-out means for publication rates and within-trial p99 values; leave-one-common-block-out ratios for all thirteen paired effects; first-five versus last-five block summaries; and summaries between recorded execution interruptions. Leave-one-out ranges measure influence, not confidence coverage. Early/late contrasts and rank correlations with block order are descriptive, with no trend significance test or causal interpretation. The interruption segments follow planned trial ranges D001–D021, D022–D030, D031–D037, D038–D042, and D043–D060, using completed primary observations only. Their unequal, small cell counts prevent treating these segments as independent controlled replications. All computations and membership lists are retained.

D.1.1 Protocol development and execution incidents

The local protocol was written before main collection but was not externally preregistered. Twelve initial pilot trials were excluded. Before main collection, we added receive-request timestamps, increased the backlog from 4096 to 16384 messages after the pilot revealed short high-concurrency runs, and added the five-second primary durability series in place of relying on prefill timings. A final twelve-trial pilot verified the revised harness. Both pilot datasets and the prospective protocol versions are retained. The execution log records these changes; they are not presented as an immutable external registration.

During collection, a stream-creation request exceeded the five-second client deadline before

a measured interval began. The server logged an 8.0186-second stream-creation handler. We retained that failed setup and the original logs, added checkpointed continuation, and retried the condition with unchanged settings and a fresh stream name. No completed measurement was replaced. After trial 88, a Redis-always warm-up returned through an error path and its cleanup also timed out, masking the primary operation error. Its original stack and log are retained. Administrative create, inventory, and deletion timeout settings were subsequently increased to 30 s. Redis default publication and acknowledgment socket timeouts and NATS response waits remained 5 s; the blocking-read overrides documented in Section E were unchanged. The durability series also stopped after 21 completed trials when a NATS-always warm-up publication timed out; the server logged approximately 4.9925 s in deletion/read-loop processing for that warm-up stream. No timed sample for its next condition had begun. We preserved the original log and all completed samples, then added append-only continuation to that series without changing its measurement path or deadlines. A further NATS-always warm-up publication timed out after 30 completed durability trials, with a corresponding 3.5595 s deletion/read-loop warning; the existing checkpointed binary then continued without a further code change. The causes of these interruptions are not established. Setup and warm-up availability are therefore reported separately from timed outcomes.

Planned timed-publication trial D038 (Redis-always) subsequently aborted with a socket read timeout. Its partial client timings and confirmed-request count were lost on process abort and cannot be reconstructed from the retained evidence. A later retained-state snapshot contained 1568 distinct application IDs and zero pending consumer acknowledgments; this cannot identify which publications the client had confirmed. We preserve the failure as the planned outcome and exclude its unavailable throughput and latency from completed-run estimates. An orchestration error during failure-ledger copying caused an unintended completed replay of D038; that replay is retained but excluded from primary statistics. Stopping that continuation also caused an operator interruption of D043, whose partial broker state was preserved and whose planned condition was rerun after cleanup. These operational deviations and the exact evidence are recorded in the artifact. No observed slow completed planned trial was removed. A plot of publication throughput by block was added as a descriptive diagnostic after observing this variability; it is not a preregistered hypothesis test.

Table 9 consolidates the 550 planned outcomes and six additional documented attempts. Its accompanying record-level inventory links each entry to retained evidence and leaves unavailable times and confirmation counts missing. Successful warm-ups are part of their completed attempts, not additional rows. The two excluded twelve-trial pilots and infrastructure provisioning are outside this inventory. This is retrospective accounting of known execution outcomes, not a prospective exclusion rule or a denominator for estimating setup availability or total workload-completion cost. In particular, the later successful D043 attempt cannot reconstruct the interrupted attempt's client history.

Retained outcome	Count	Treatment in this study
Completed timing trials	419	300 drain, 59 publication, 60 serial
Active consumer recovery	120	Recovered identifiers and cleared pending state
No-reclaimer controls	10	Pending at reconciliation; nominal budgets only
Planned publication failure	1	D038; client timings unavailable
Additional setup failure	1	Tmain052; excluded from timed estimates
Additional warm-up failures	3	Tmain089, D022, D031; timed trials not begun
Operator interruption	1	D043; interrupted client trace unavailable
Completed diagnostic replay	1	D038; retained and excluded from estimates

Table 9: Retained campaign outcomes and additional documented attempts. The first four rows account for all 550 planned outcomes. The remaining six attempts are separate from those outcomes. Counts do not reconstruct missing durations or establish end-to-end availability.

D.2 Original campaign results

D.2.1 Recorded outcomes and evidence quality

All 300 planned backlog-drain trials completed and their 4,915,200 message records passed complete identifier-set, uniqueness, retained-count, and pending-state checks. The 60 serial sensitivity trials similarly reconciled 61,440 messages. The primary publication series has 59 completed planned trials with 12,434,648 confirmed appends and one retained Redis-always timeout. Its accidental completed replay is excluded as specified in Section 4. Publication-only integrity checks establish the retained count, not a complete payload readback. The analyzer reconstructs each reported throughput and percentile from raw timings and rejects inconsistent summaries or a missing failure ledger.

All 120 active recovery episodes returned the original identifier, acknowledged it, and left one retained message with zero pending acknowledgments; JetStream reported two deliveries in each episode. All ten Redis no-reclaimer controls remained pending at the end of their observation windows. Every victim exit was recorded as killed by a signal. The largest kill-invocation-to-observed-exit bracket was 2.291 ms. The final seven-pod snapshot reported zero container restarts and zero cgroup OOM kills. Recorded driver CPU-counter differences showed no throttling within completed timed trials; this does not establish an idle testbed or identify the cause of the recorded timeouts.

D.2.2 RQ1: failure age, reclamation, and observed redelivery

Table 10 and Figure 3 report shorter failure-to-redelivery intervals after later kills at the same configured timeout. At $\Delta = 250$ ms, the nominal remaining intervals are 225 and 62.5 ms. JetStream's corresponding medians are 226.5 and 63.6 ms; Redis with a 10 ms reclaim sleep yields 230.2 and 67.9 ms. At $\Delta = 1000$ ms, the medians decrease from 907.1 to 257.9 ms for JetStream and from 903.5 to 255.4 ms for Redis with the shorter sleep. This direction is consistent with the residual-time decomposition in Theorem C.1: later failure leaves less of the existing eligibility interval to wait out. Eligibility depends on message age and acknowledgment state; it can arise while a consumer remains alive. These observations do not establish a death-detection mechanism

Δ	α	System / sleep	$\Delta(1-\alpha)$ (ms)	Median (ms)	Range (ms)
250	0.1	JetStream	225.0	226.5	[225.9, 226.9]
250	0.1	Redis / 10 ms	225.0	230.2	[228.4, 234.3]
250	0.1	Redis / 100 ms	225.0	265.4	[234.0, 313.9]
250	0.75	JetStream	62.5	63.6	[63.1, 64.5]
250	0.75	Redis / 10 ms	62.5	67.9	[63.9, 70.6]
250	0.75	Redis / 100 ms	62.5	134.0	[75.8, 161.2]
1000	0.1	JetStream	900.0	907.1	[905.4, 910.2]
1000	0.1	Redis / 10 ms	900.0	903.5	[900.4, 913.7]
1000	0.1	Redis / 100 ms	900.0	943.5	[911.7, 981.1]
1000	0.75	JetStream	250.0	257.9	[255.5, 260.6]
1000	0.75	Redis / 10 ms	250.0	255.4	[251.2, 269.4]
1000	0.75	Redis / 100 ms	250.0	319.4	[258.9, 339.4]

Table 10: Failure-to-redelivery latency for all recovery configurations. Each row contains ten killed-consumer episodes. $\Delta(1-\alpha)$ is a nominal reference based on the scheduled age after client receipt, not a verified server-side lower bound. Ranges contain the minimum and maximum observed episode.

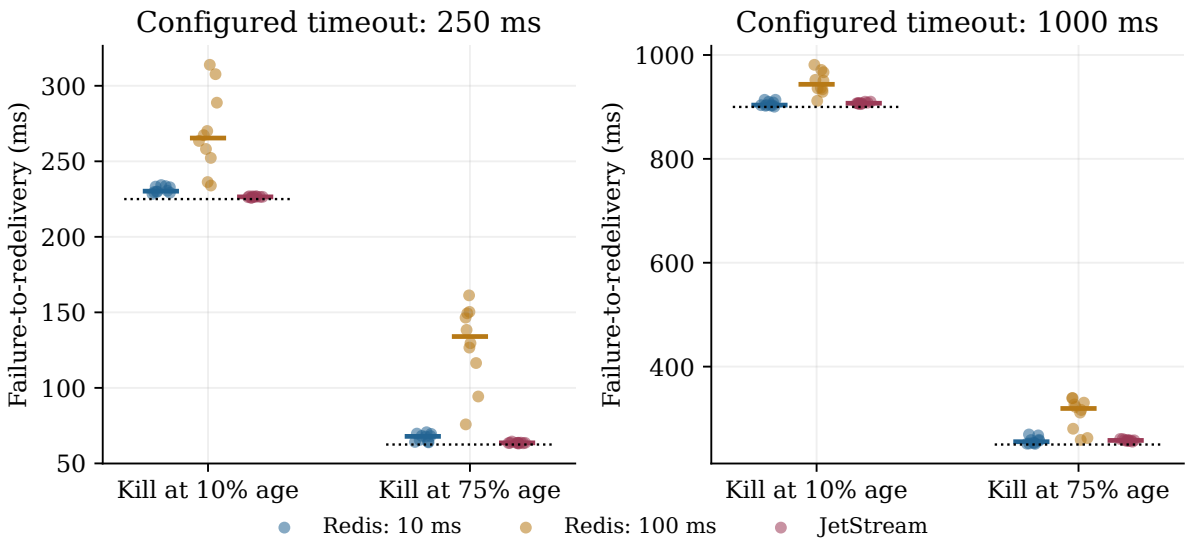

Figure 3: Every active-reclamation recovery observation. Horizontal colored marks show medians; horizontal jitter separates points and has no numerical meaning. Dotted lines show the nominal client-receipt-based residual-time reference. The ten no-reclaimer controls are censored and are reported separately rather than plotted as exact recovery times.

or directly observe the broker-side timer origin.

Redis reclamation frequency also matters. With a 100 ms sleep, the late-failure median rises to 134.0 ms at the 250 ms timeout and 319.4 ms at the 1000 ms timeout. Across the four age/timeout cells, the median observed excess $(\widehat{r} - \widehat{d}) - \Delta$ ranges from 4.0 to 6.2 ms with a 10 ms sleep, compared with 40.7–72.2 ms with a 100 ms sleep. JetStream's median excess ranges from 1.9 to 8.6 ms. Neither a configured sleep nor an observed maximum is a proven future scheduling bound. JetStream's 1000 ms medians are slightly above the shorter-sleep Redis medians, so the results do not support an unconditional claim that one system recovers faster.

The negative controls isolate a separate mechanism: ten observed non-recoveries with pending state intact are consistent with Proposition C.3. Their finite observation procedures do not empirically prove infinite non-delivery; the proposition supplies that conditional counterexample from the stated new-read-only policy. Because exact censor endpoints were not recorded, these controls contribute descriptive evidence rather than exposure-time or survival estimates. The answer to RQ1 is therefore twofold: a failure leaves a residual timeout, and actual recovery additionally depends on reclamation or expiry selection and available demand. Under the active policies studied, all episodes recovered; without Redis reclamation, none of the controls did during observation.

D.2.3 RQ2: concurrency gains and cycle-latency cost

System	Mode	X_1 (kmsg/s)	X_{16} (kmsg/s)	X_{16}/X_1
Redis	memory	2.89 [2.74, 3.01]	36.37 [35.94, 36.83]	12.59 [12.13, 13.15]
Redis	periodic	2.66 [2.56, 2.76]	29.32 [27.73, 30.50]	11.02 [10.22, 11.79]
Redis	always	0.29 [0.29, 0.30]	2.51 [2.47, 2.55]	8.59 [8.47, 8.72]
JetStream	memory	2.30 [2.26, 2.34]	29.79 [29.43, 30.12]	12.94 [12.69, 13.23]
JetStream	periodic	2.27 [2.24, 2.29]	28.03 [26.89, 28.97]	12.36 [11.79, 12.83]
JetStream	always	2.24 [2.21, 2.27]	27.43 [27.04, 27.83]	12.24 [11.98, 12.54]

Table 11: Acknowledged backlog-drain throughput at one and sixteen consumers, with means and nominal 95% bootstrap intervals. Speedup is the ratio of condition means with a paired block-bootstrap interval. Each condition contains ten trials.

Increasing C from one to sixteen improved mean drain throughput in every profile (Table 11). Redis gains were 12.59, 11.02, and 8.59 times for memory, periodic, and always modes; JetStream gains were 12.94, 12.36, and 12.24 times. All six endpoint gains are below the sixteenfold increase in workers. The memory profiles reached 36.37 kmsg/s for Redis and 29.79 kmsg/s for JetStream at sixteen consumers. Periodic profiles reached 29.32 and 28.03 kmsg/s. These rates describe the measured request/acknowledgment path with a prefilled backlog, not a server-capacity ceiling or an open-loop saturation point.

The throughput gain has a cycle-latency cost. From one to sixteen consumers, mean within-trial cycle p99 increased from 0.543 to 0.867 ms for Redis memory and from 0.653 to 1.310 ms for JetStream memory. Redis periodic increased from 0.565 to 1.334 ms and JetStream periodic from 0.636 to 1.520 ms. Redis always increased from 5.942 to 10.311 ms, while JetStream always increased from 0.636 to 1.541 ms. In contrast, the corresponding common-start-to-receipt drain p99 values decrease between these endpoints (the complete matrix below): more workers complete the backlog sooner despite a slower individual cycle.

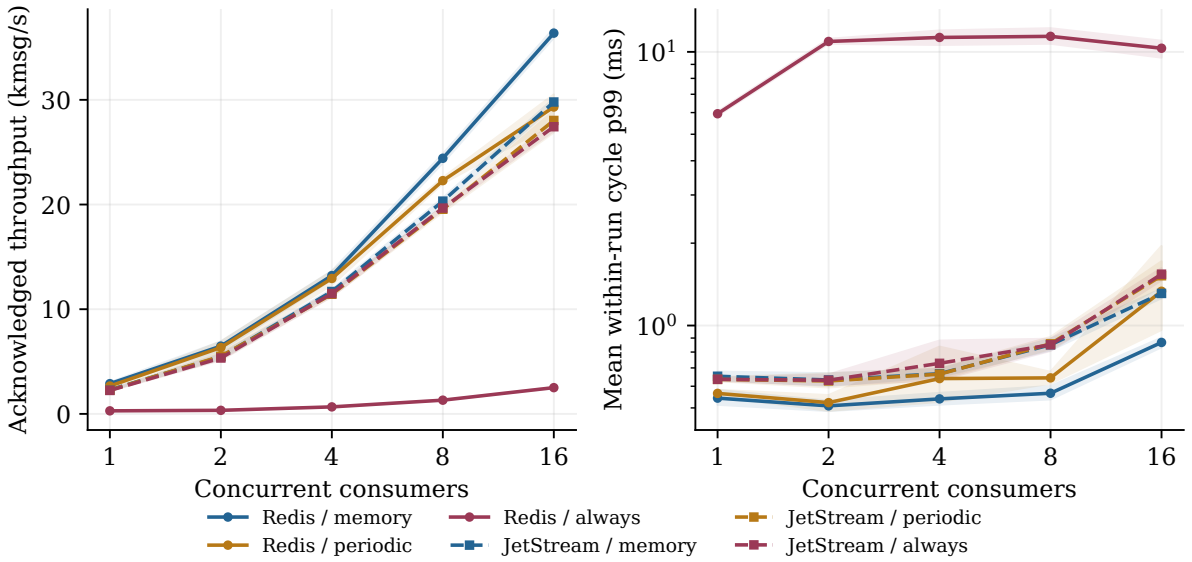

Figure 4: The complete concurrency response. Lines connect measured consumer counts without estimating unmeasured counts. Shaded bands are nominal 95% bootstrap confidence intervals for condition means; the right panel uses a logarithmic latency axis. Cycle latency is successful receive-request invocation to acknowledgment completion.

Reconstructed occupancy U lies between 0.9936 and 0.9995 across all 300 trials. Every worker's recorded intervals are nonoverlapping and inside the measured window, satisfying the checked premises of Theorem C.5. Thus recorded successful cycles occupy nearly all available worker time. The identity $X = CU/\overline{R}$ explains how increased mean cycle cost offsets the added concurrency; it does not identify which internal contention mechanism caused that cost. Nor does a p99 value replace $\overline{R}$ in this identity.

The always-mode endpoint is particularly easy to misread: JetStream drains 27.43 kmsg/s and Redis 2.51 kmsg/s, but their consumer-state synchronization contracts differ. RQ2 is answered by sublinear endpoint scaling with rising cycle tails within each profile. A cross-system ranking under an equal synchronous consumer-progress durability guarantee is not established by these configurations.

D.2.4 RQ3: confirmed-publication cost under persistence policies

The periodic configurations exhibited lower confirmed-publication throughput than the memory configurations in this campaign. Memory and periodic means were 73.90 and 61.38 kmsg/s for Redis, and 60.38 and 49.49 kmsg/s for JetStream. The paired periodic/memory throughput ratios were 0.8305 [0.8237, 0.8371] for Redis and 0.8197 [0.7933, 0.8511] for JetStream, using nominal 95% block-bootstrap intervals. Mean within-trial publication p99 was 0.474 versus 0.557 ms and 0.704 versus 0.849 ms, respectively. These descriptive differences include the evolving storage history and shared environment; they do not isolate the causal cost of synchronization policy or measure power-loss data loss.

Always-mode publication was substantially slower and more variable. The nine completed planned Redis-always trials averaged 3028 msg/s [1725, 4306], with a mean within-trial p99 of

System	Mode	n	Throughput (kmsg/s)	p50 (ms)	p99 (ms)
Redis	memory	10	73.90 [73.60, 74.19]	0.204	0.474 [0.466, 0.486]
Redis	periodic	10	61.38 [61.08, 61.67]	0.247	0.557 [0.551, 0.565]
Redis	always	9	3.03 [1.73, 4.31]	9.959	29.001 [11.615, 50.929]
JetStream	memory	10	60.38 [58.99, 61.32]	0.241	0.704 [0.678, 0.742]
JetStream	periodic	10	49.49 [48.29, 50.48]	0.281	0.849 [0.832, 0.869]
JetStream	always	10	0.37 [0.19, 0.56]	108.664	579.568 [91.362, 1,459.647]

Table 12: Five-second publication workload with sixteen independent publishers. Throughput and p99 entries give means and nominal 95% bootstrap intervals; p50 entries are means of within-trial medians. n is the number of completed planned trials out of ten; the failed Redis-always trial is reported separately.

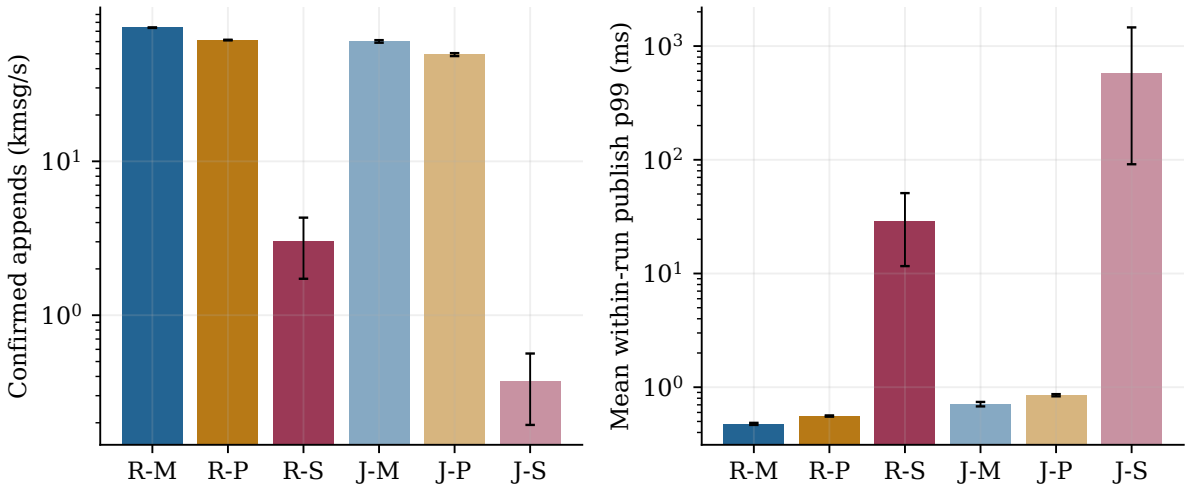

Figure 5: Confirmed-publication throughput and latency for Redis (R) and JetStream (J), in memory (M), periodic (P), and always (S) profiles. Both vertical axes are logarithmic. Error bars give nominal 95% bootstrap confidence intervals for means conditional on completion (nine Redis-always trials and ten for each other profile).

29.0 ms [11.6, 50.9]. The tenth planned trial timed out at the measured request deadline and has no reconstructible client latency sample. The ten completed JetStream-always trials averaged 375 msg/s [194, 564], with a mean within-trial p99 of 579.6 ms [91.4, 1459.6]. The wide intervals and per-block observations in Figure 6 are material results: the point estimates should not be presented as stable production tail-latency guarantees. The separately recorded setup and warm-up incidents further constrain operational interpretation.

System	Mode	Mean p99 (ms)	Median p99 (ms)	LOO means (ms)	Early / late rate (kmsg/s)
Redis	memory	0.47	0.47	[0.47, 0.48]	73.98 / 73.83
Redis	periodic	0.56	0.56	[0.56, 0.56]	61.38 / 61.37
Redis	always	29.00	7.46	[20.15, 32.10]	3.78 / 2.08
JetStream	memory	0.70	0.68	[0.69, 0.71]	59.68 / 61.07
JetStream	periodic	0.85	0.84	[0.84, 0.85]	50.35 / 48.64
JetStream	always	579.57	160.26	[150.53, 639.37]	0.45 / 0.30

Table 13: Post-observation publication sensitivity. The mean and median summarize completed trials' p99 values. LOO gives the range of means after omitting each completed trial in turn, solely as an influence diagnostic. Early and late rates use blocks 1–5 and 6–10. Redis-always has five early and four late completions; other cells have five in each half. No observation is removed from primary estimates.

Table 13 shows the sensitivity of always-mode summaries. JetStream-always's mean within-trial p99 is 579.6 ms, whereas the median of those ten p99 values is 160.3 ms. Its leave-one-out means range from 150.5 to 639.4 ms; the minimum omits D040, whose within-trial p99 is 4440.9 ms. Redis-always's corresponding mean and median are 29.0 and 7.5 ms. These contrasts characterize influence and skew, not grounds for discarding slow observations.

System	Mode	Trials	Messages per trial	Positions above p99
Redis	memory	10	365,505–372,128	3,655–3,721
Redis	periodic	10	303,669–310,204	3,036–3,102
Redis	always	9	2,608–26,221	26–262
JetStream	memory	10	273,779–310,872	2,737–3,108
JetStream	periodic	10	243,055–257,750	2,430–2,577
JetStream	always	10	149–3,577	1–35

Table 14: Sample information underlying completed publication trials' empirical p99 values. Ranges span the retained trials of each profile. Positions above p99 are $N - \lceil 0.99N \rceil$ and do not imply independent tail observations.

Table 14 makes the unequal sample sizes explicit. All 59 trial counts, ranks, and p99 values are retained in `data/derived/publication-tail-samples.csv`. D040 contains 149 confirmed publications, so its p99 is order statistic 148, the second-largest observation. Its influential tail estimate consequently has very little information above the selected rank. The mean of trial p99 values remains a descriptive summary of these observed trials; it is weak evidence for a stable population tail, and equal trial weights do not equalize percentile precision. Longer, independently repeated observations would be required for a stronger tail characterization. We retain D040 and do not replace its percentile with a pooled-message estimate.

The direction of all thirteen reported paired contrasts survives every leave-one-common-block-out calculation. Magnitudes remain conditional on the retained blocks: the periodic/memory publication ratios range from 0.8285 to 0.8320 for Redis and from 0.8076 to 0.8282 for JetStream in that diagnostic. Early versus late always-mode throughput declines from 3.78 to 2.08 kmsg/s for Redis and from 0.45 to 0.30 kmsg/s for JetStream. A simple monotonic degradation model is also insufficient: JetStream-always averages 483.3 msg/s in D001–D021, 24.6 msg/s in the interruption-bounded D038–D042 segment containing its single D040 observation, and 462.7 msg/s in D043–D060. The artifact reports every segment and cell count. These uncontrolled contrasts establish variability and sensitivity to execution history, not that an interruption or AOF growth caused a particular outcome. Independent campaign repetitions are needed to assess repeatability beyond this deployment.

Against memory mode, the paired always/memory throughput ratio is 0.0410 [0.0233, 0.0583] for Redis over nine common completed blocks and 0.00621 [0.00318, 0.00945] for JetStream over ten blocks. Their reciprocal point estimates are 24.4- and 161.1-fold slowdowns. Because the Redis estimate excludes an unavailable failed-trial metric, it describes successful runs and can understate the operational cost of the complete outcome distribution. We neither impute zero throughput nor replace the failed outcome with its diagnostic replay.

The original serial-publisher sensitivity gives a different view (Appendix B): always-mode publication averaged 603 msg/s [591, 614] for Redis and 585 msg/s [565, 598] for JetStream. The close point estimates do not establish equivalence. This shorter fixed-count workload differs in both publication concurrency and duration, so it cannot isolate batching as the cause of the contrast with the sixteen-publisher results. No syscall or storage-device trace was collected in that original campaign to establish the mechanism. The follow-up's matched-duration publication and syscall diagnostics are a separate dataset.

RQ3 is therefore answered as a descriptive, workload-specific tradeoff: the configured periodic profiles preserve most memory-mode throughput with modestly higher observed tails, whereas always-mode confirmed publication incurs much larger and more variable observed costs in this shared Btrfs/NVMe environment. The synchronization contract, accumulated storage history, completion conditioning, and failure class must accompany that performance statement. Neither these append measurements nor the fast JetStream drain acknowledgments prove equivalent durable consumer progress or stable-media survival after a power cut.

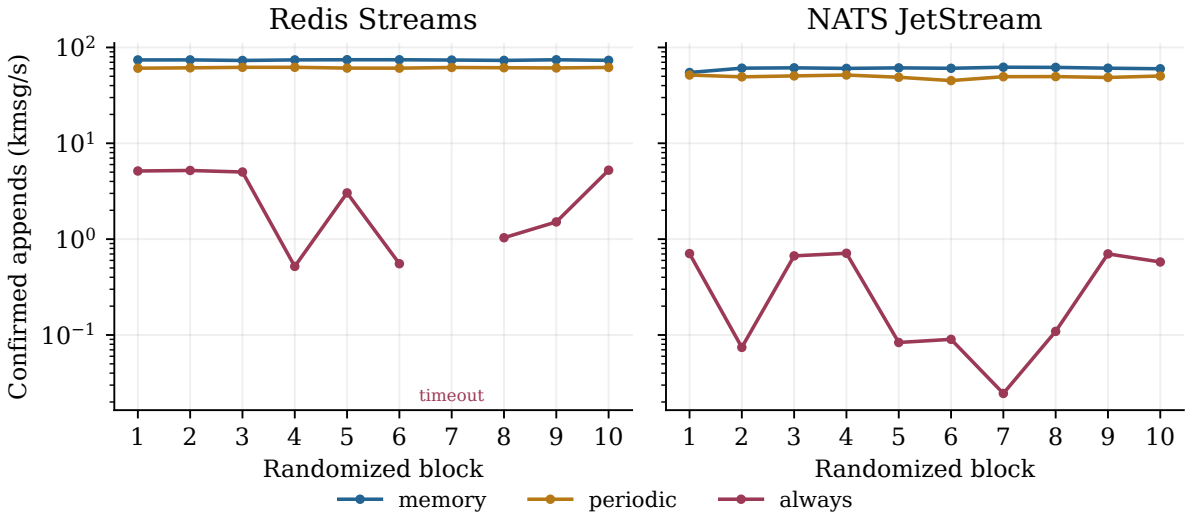

Figure 6: Descriptive publication-throughput diagnostic by randomized block. Each point is one completed planned trial. The timeout annotation has no throughput coordinate and identifies the failed Redis-always trial in block seven. A missing value interrupts its line; no failure is imputed as zero. Blocks are ordered experimental groups, not equal wall-time intervals.

E Client waits and observation conventions

All request timestamps use the driver's Linux monotonic clock. Broker telemetry uses its own node clock. Setup, warm-up, trace serialization and stored-payload readback are outside timing. Original ordinary reads and recovery survivor polls use a 100 ms Redis block or JetStream fetch wait. Follow-up drain requests and victim initial reads use five seconds. The pinned Redis client adds ten seconds to positive typed blocking-read socket deadlines, producing 10.1 s or 15 s, respectively [30]; the raw CLAIM command instead retains the default five-second socket timeout. JetStream's fetch-request expiry is approximately 90% of the client wait and can leave a pull-demand gap [37]. Redis publication and acknowledgment use five-second socket-stage read/write limits, not a whole-call deadline; NATS uses five-second response waits. The original drain loop additionally checks a 120 s budget, while the external follow-up invocation has a separate 600 s controller limit. None of these waits is the broker eligibility threshold.

F Complete follow-up results

The following tables concern the separately collected fresh-store follow-up. Runs 01, 02 and 03 identify separate deployments on the same hardware. Cross-run point summaries equally weight the available run estimates. Bracketed ranges contain their minimum and maximum; they are not confidence or prediction intervals. M, P and S denote memory, periodic and always append profiles. Publication P is the number of publishers, whereas drain C is the number of consumers. A failed outcome is absent from successful-run numerical means but remains in the outcome and pair counts.

F.1 Publication means and tail sample information

Rates are in kmsg/s. Trial p99 is the equally weighted mean of within-trial nearest-rank percentiles inside each run, followed by equal weighting across available run means. n is completed/planned trials. The range of sample counts is reported by run below; every trial's count, selected p99 rank and number of positions above that rank are in the accompanying `followup/derived/trials.csv`.

System	Mode	P	n	Rate, kmsg/s	Trial p99, ms
Redis 7.4.2	M	1	9/9	5.73 [5.50, 5.86]	0.26 [0.25, 0.26]
Redis 7.4.2	M	16	9/9	72.13 [71.34, 72.62]	0.49 [0.49, 0.50]
Redis 7.4.2	P	1	9/9	5.26 [4.87, 5.55]	0.27 [0.26, 0.29]
Redis 7.4.2	P	16	9/9	60.70 [60.20, 61.21]	0.56 [0.56, 0.56]
Redis 7.4.2	S	1	7/9	0.40 [0.35, 0.42]	16.73 [13.95, 20.20]
Redis 7.4.2	S	16	7/9	4.51 [3.83, 5.00]	11.78 [6.16, 18.90]
Redis 8.10.2	M	1	9/9	5.50 [5.48, 5.52]	0.26 [0.26, 0.26]
Redis 8.10.2	M	16	9/9	73.26 [72.80, 73.71]	0.47 [0.46, 0.48]
Redis 8.10.2	P	1	9/9	5.36 [5.30, 5.47]	0.27 [0.26, 0.27]
Redis 8.10.2	P	16	9/9	60.49 [59.57, 61.13]	0.56 [0.55, 0.58]
Redis 8.10.2	S	1	6/9	0.51 [0.35, 0.59]	7.72 [2.68, 17.13]
Redis 8.10.2	S	16	9/9	4.35 [3.46, 5.05]	16.98 [6.02, 29.12]
JetStream 2.10.24	M	1	9/9	4.61 [4.59, 4.61]	0.31 [0.31, 0.32]
JetStream 2.10.24	M	16	9/9	60.47 [59.79, 61.13]	0.69 [0.67, 0.71]
JetStream 2.10.24	P	1	8/9	4.19 [4.11, 4.29]	0.35 [0.34, 0.36]
JetStream 2.10.24	P	16	9/9	49.16 [47.77, 50.34]	0.90 [0.86, 0.94]
JetStream 2.10.24	S	1	9/9	0.44 [0.36, 0.54]	13.38 [4.40, 21.93]
JetStream 2.10.24	S	16	8/9	0.55 [0.46, 0.63]	105.96 [48.88, 165.48]
JetStream 2.15.0	M	1	9/9	4.48 [4.47, 4.50]	0.32 [0.32, 0.32]
JetStream 2.15.0	M	16	9/9	60.62 [60.36, 60.84]	0.64 [0.63, 0.65]
JetStream 2.15.0	P	1	9/9	4.22 [4.20, 4.23]	0.34 [0.33, 0.34]
JetStream 2.15.0	P	16	9/9	49.75 [49.09, 50.09]	0.80 [0.78, 0.82]
JetStream 2.15.0	S	1	8/9	0.46 [0.41, 0.57]	12.07 [2.78, 22.97]
JetStream 2.15.0	S	16	8/9	0.56 [0.42, 0.68]	295.64 [45.58, 725.22]

F.2 Publication by deployment

Each condition has three planned trials within a deployment. Columns report the condition's arithmetic mean throughput and mean empirical trial p99, conditional on completion. Count range is the minimum/maximum number of confirmed publications among those completed trials.

Run	System	Mode	P	n	kmsg/s	p99, ms	Count range
01	Redis 7.4.2	M	1	3/3	5.86	0.26	86617–89404
02	Redis 7.4.2	M	1	3/3	5.83	0.25	86695–88585
03	Redis 7.4.2	M	1	3/3	5.50	0.26	73168–87462
01	Redis 7.4.2	M	16	3/3	72.42	0.49	1063608–1103486
02	Redis 7.4.2	M	16	3/3	71.34	0.50	1009276–1102191
03	Redis 7.4.2	M	16	3/3	72.62	0.49	1069841–1101776
01	Redis 7.4.2	P	1	3/3	5.55	0.26	82805–83646
02	Redis 7.4.2	P	1	3/3	5.37	0.27	73441–84247
03	Redis 7.4.2	P	1	3/3	4.87	0.29	72885–73175
01	Redis 7.4.2	P	16	3/3	60.20	0.56	875200–924144
02	Redis 7.4.2	P	16	3/3	60.69	0.56	887260–927302
03	Redis 7.4.2	P	16	3/3	61.21	0.56	897989–933609
01	Redis 7.4.2	S	1	3/3	0.42	13.95	886–9143
02	Redis 7.4.2	S	1	2/3	0.35	20.20	1761–8773

Run	System	Mode	P	n	kmsg/s	p99, ms	Count range
03	Redis 7.4.2	S	1	2/3	0.42	16.05	3585–8903
01	Redis 7.4.2	S	16	2/3	3.83	18.90	37311–77608
02	Redis 7.4.2	S	16	2/3	5.00	6.16	73784–76364
03	Redis 7.4.2	S	16	3/3	4.69	10.29	59595–76581
01	Redis 8.10.2	M	1	3/3	5.52	0.26	74269–87818
02	Redis 8.10.2	M	1	3/3	5.48	0.26	73464–87073
03	Redis 8.10.2	M	1	3/3	5.50	0.26	73999–87022
01	Redis 8.10.2	M	16	3/3	72.80	0.48	1084779–1101455
02	Redis 8.10.2	M	16	3/3	73.71	0.46	1099413–1109992
03	Redis 8.10.2	M	16	3/3	73.28	0.46	1095054–1105853
01	Redis 8.10.2	P	1	3/3	5.30	0.27	72271–84374
02	Redis 8.10.2	P	1	3/3	5.32	0.27	73471–83211
03	Redis 8.10.2	P	1	3/3	5.47	0.26	81896–82151
01	Redis 8.10.2	P	16	3/3	61.13	0.55	910897–923491
02	Redis 8.10.2	P	16	3/3	59.57	0.58	847181–922708
03	Redis 8.10.2	P	16	3/3	60.76	0.55	895930–926585
01	Redis 8.10.2	S	1	2/3	0.59	3.35	8675–8929
02	Redis 8.10.2	S	1	2/3	0.59	2.68	8815–8870
03	Redis 8.10.2	S	1	2/3	0.35	17.13	1685–8858
01	Redis 8.10.2	S	16	3/3	5.05	6.02	73322–77497
02	Redis 8.10.2	S	16	3/3	3.46	29.12	30549–76566
03	Redis 8.10.2	S	16	3/3	4.52	15.81	59508–76001
01	JetStream 2.10.24	M	1	3/3	4.59	0.32	68143–69356
02	JetStream 2.10.24	M	1	3/3	4.61	0.31	68542–70456
03	JetStream 2.10.24	M	1	3/3	4.61	0.31	68729–69462
01	JetStream 2.10.24	M	16	3/3	59.79	0.71	886488–909818
02	JetStream 2.10.24	M	16	3/3	60.51	0.69	894937–928256
03	JetStream 2.10.24	M	16	3/3	61.13	0.67	914429–920304
01	JetStream 2.10.24	P	1	3/3	4.11	0.36	58828–63146
02	JetStream 2.10.24	P	1	3/3	4.29	0.34	63721–65551
03	JetStream 2.10.24	P	1	2/3	4.17	0.35	61715–63332
01	JetStream 2.10.24	P	16	3/3	49.36	0.88	733595–745935
02	JetStream 2.10.24	P	16	3/3	47.77	0.94	677445–738843
03	JetStream 2.10.24	P	16	3/3	50.34	0.86	747495–764302
01	JetStream 2.10.24	S	1	3/3	0.42	13.81	1510–8602
02	JetStream 2.10.24	S	1	3/3	0.36	21.93	1022–8572
03	JetStream 2.10.24	S	1	3/3	0.54	4.40	7674–8753
01	JetStream 2.10.24	S	16	3/3	0.55	103.53	3941–10488
02	JetStream 2.10.24	S	16	2/3	0.46	165.48	3535–10366
03	JetStream 2.10.24	S	16	3/3	0.63	48.88	8839–10358
01	JetStream 2.15.0	M	1	3/3	4.47	0.32	64320–68841
02	JetStream 2.15.0	M	1	3/3	4.47	0.32	64159–69257
03	JetStream 2.15.0	M	1	3/3	4.50	0.32	64403–69803
01	JetStream 2.15.0	M	16	3/3	60.84	0.64	906793–918179
02	JetStream 2.15.0	M	16	3/3	60.36	0.65	876947–928091
03	JetStream 2.15.0	M	16	3/3	60.67	0.63	905314–917583
01	JetStream 2.15.0	P	1	3/3	4.20	0.34	62006–63665
02	JetStream 2.15.0	P	1	3/3	4.23	0.33	62657–64606
03	JetStream 2.15.0	P	1	3/3	4.23	0.34	62445–64079
01	JetStream 2.15.0	P	16	3/3	50.09	0.79	738387–768069
02	JetStream 2.15.0	P	16	3/3	50.07	0.78	744559–755619
03	JetStream 2.15.0	P	16	3/3	49.09	0.82	717729–764826
01	JetStream 2.15.0	S	1	2/3	0.42	10.45	4135–8392
02	JetStream 2.15.0	S	1	3/3	0.41	22.97	1022–8614
03	JetStream 2.15.0	S	1	3/3	0.57	2.78	8546–8603
01	JetStream 2.15.0	S	16	2/3	0.68	45.58	10169–10236

Run	System	Mode	P	n	kmsg/s	p99, ms	Count range
02	JetStream 2.15.0	S	16	3/3	0.60	116.10	7895–10276
03	JetStream 2.15.0	S	16	3/3	0.42	725.22	976–10394

F.3 Current versus historical releases

Each entry is the equal-weight mean of the three deployment-specific paired throughput ratios, followed by their observed range. Ratios use jointly completed blocks under the same profile and publisher count. Redis compares 8.10.2 with 7.4.2; JetStream compares 2.15.0 with 2.10.24. Pairs gives available/planned matched blocks. The intervals are descriptive ranges, and proximity to one is not an equivalence test.

System	Profile	P	Current/historical	Pairs
Redis	Memory	1	0.9603 [0.9401, 0.9998]	9/9
Redis	Memory	16	1.0158 [1.0052, 1.0332]	9/9
Redis	Periodic	1	1.0232 [0.9544, 1.1237]	9/9
Redis	Periodic	16	0.9966 [0.9816, 1.0155]	9/9
JetStream	Memory	1	0.9728 [0.9693, 0.9761]	9/9
JetStream	Memory	16	1.0025 [0.9925, 1.0177]	9/9
JetStream	Periodic	1	1.0066 [0.9869, 1.0211]	8/9
JetStream	Periodic	16	1.0127 [0.9750, 1.0482]	9/9

Table 17: Current/historical publication throughput ratios for memory and periodic profiles.

F.4 Drain by deployment

Each condition has two planned trials per deployment and 65536 messages per completed trial. CPU is the driver quota, with matching GOMAXPROCS. Driver and Broker columns give observed mean cores during their phase-bracketing samples, including other processes within the respective containers. The p99 column is the mean empirical request-to-acknowledgment cycle p99 in milliseconds.

Run	System	Mode	C	CPU	n	kmsg/s	p99, ms	Driver	Broker
01	Redis	M	1	2	2/2	2.70	0.53	0.33	0.08
02	Redis	M	1	2	2/2	2.90	0.50	0.36	0.08
03	Redis	M	1	2	2/2	2.90	0.51	0.36	0.09
01	Redis	M	16	2	2/2	34.74	0.90	1.07	0.56
02	Redis	M	16	2	2/2	35.36	0.88	1.08	0.56
03	Redis	M	16	2	2/2	34.85	0.90	1.08	0.56
01	Redis	M	1	4	2/2	2.93	0.51	0.35	0.08
02	Redis	M	1	4	2/2	2.69	0.53	0.34	0.08
03	Redis	M	1	4	2/2	2.92	0.50	0.36	0.08
01	Redis	M	16	4	2/2	35.77	0.81	1.42	0.56
02	Redis	M	16	4	2/2	36.39	0.76	1.44	0.56
03	Redis	M	16	4	2/2	35.99	0.82	1.43	0.56
01	Redis	P	1	2	2/2	2.45	0.56	0.32	0.12
02	Redis	P	1	2	2/2	2.59	0.54	0.33	0.12
03	Redis	P	1	2	2/2	2.73	0.53	0.35	0.13
01	Redis	P	16	2	2/2	28.69	1.04	0.98	0.65
02	Redis	P	16	2	2/2	28.46	1.04	0.97	0.64

Run	System	Mode	C	CPU	n	kmsg/s	p99, ms	Driver	Broker
03	Redis	P	16	2	2/2	28.51	1.04	0.98	0.64
01	Redis	P	1	4	2/2	2.45	0.56	0.31	0.12
02	Redis	P	1	4	2/2	2.60	0.54	0.33	0.12
03	Redis	P	1	4	2/2	2.61	0.54	0.33	0.12
01	Redis	P	16	4	2/2	28.49	0.94	1.31	0.65
02	Redis	P	16	4	2/2	29.84	0.90	1.32	0.64
03	Redis	P	16	4	2/2	29.59	0.98	1.33	0.63
01	JetStream	M	1	2	2/2	2.28	0.62	0.45	0.22
02	JetStream	M	1	2	2/2	2.29	0.62	0.45	0.22
03	JetStream	M	1	2	2/2	2.25	0.64	0.43	0.21
01	JetStream	M	16	2	2/2	29.65	1.24	1.11	1.30
02	JetStream	M	16	2	2/2	30.56	1.17	1.12	1.30
03	JetStream	M	16	2	2/2	28.41	1.42	1.10	1.27
01	JetStream	M	1	4	2/2	2.31	0.62	0.45	0.22
02	JetStream	M	1	4	2/2	2.30	0.63	0.45	0.22
03	JetStream	M	1	4	2/2	2.24	0.63	0.45	0.21
01	JetStream	M	16	4	2/2	32.71	0.97	1.50	1.36
02	JetStream	M	16	4	2/2	32.90	0.98	1.50	1.34
03	JetStream	M	16	4	2/2	32.40	1.01	1.48	1.34
01	JetStream	P	1	2	2/2	2.28	0.63	0.44	0.22
02	JetStream	P	1	2	2/2	2.27	0.64	0.44	0.22
03	JetStream	P	1	2	2/2	2.22	0.65	0.44	0.22
01	JetStream	P	16	2	2/2	28.78	1.32	1.11	1.29
02	JetStream	P	16	2	2/2	28.95	1.31	1.12	1.28
03	JetStream	P	16	2	2/2	28.37	1.42	1.10	1.27
01	JetStream	P	1	4	2/2	2.26	0.64	0.45	0.22
02	JetStream	P	1	4	2/2	2.28	0.64	0.45	0.22
03	JetStream	P	1	4	2/2	2.28	0.62	0.45	0.22
01	JetStream	P	16	4	2/2	31.67	1.07	1.47	1.35
02	JetStream	P	16	4	2/2	31.57	1.07	1.48	1.34
03	JetStream	P	16	4	2/2	31.61	1.09	1.46	1.33

F.5 Recovery excess and paired controls

Excess is redelivery receipt minus initial receipt minus the configured threshold. Each live/killed cell has two planned episodes in each deployment. Entries give equally weighted deployment means and their observed ranges. Killed minus live is calculated within paired blocks before averaging deployment estimates. R7/R8 denote Redis 7.4.2/8.10.2, auto denotes XAUTOCLAIM with a 10 ms sleep, and J denotes JetStream. Neither a small observed difference nor overlap establishes equivalence.

Policy	Δ	Age, %	Live excess, ms	Killed excess, ms	Killed − live, ms
R7 auto	250	10	6.38 [5.04, 8.82]	5.61 [0.98, 8.09]	-0.77 [-4.06, 2.48]
R8 auto	250	10	5.44 [2.70, 7.98]	6.05 [5.17, 6.99]	0.61 [-0.99, 3.28]
R8 CLAIM	250	10	80.02 [78.43, 81.20]	78.29 [74.45, 81.22]	-1.74 [-6.75, 2.78]
J2.10	250	10	2.12 [2.05, 2.25]	2.17 [2.09, 2.21]	0.05 [-0.16, 0.16]
J2.15	250	10	1.98 [1.62, 2.17]	1.94 [1.71, 2.22]	-0.04 [-0.47, 0.28]
R7 auto	250	75	5.11 [3.73, 7.13]	7.25 [3.30, 10.50]	2.14 [-1.17, 6.77]
R8 auto	250	75	6.66 [2.84, 9.21]	2.26 [1.60, 3.52]	-4.40 [-7.55, 0.68]
R8 CLAIM	250	75	72.12 [62.22, 78.60]	81.08 [76.86, 87.83]	8.96 [3.02, 14.64]
J2.10	250	75	2.02 [1.97, 2.10]	2.00 [1.95, 2.07]	-0.02 [-0.12, 0.09]
J2.15	250	75	1.93 [1.63, 2.23]	2.01 [1.90, 2.10]	0.08 [-0.33, 0.46]
R7 auto	1000	10	8.71 [8.13, 9.53]	4.66 [2.80, 6.13]	-4.05 [-6.72, -2.00]
R8 auto	1000	10	8.08 [6.00, 10.70]	6.30 [3.20, 10.52]	-1.78 [-5.52, 4.52]

Policy	Δ	Age, %	Live excess, ms	Killed excess, ms	Killed − live, ms
R8 CLAIM	1000	10	35.63 [26.64, 53.40]	30.04 [26.49, 36.71]	-5.58 [-26.91, 9.87]
J2.10	1000	10	7.17 [6.28, 7.87]	8.57 [8.21, 8.81]	1.40 [0.34, 2.53]
J2.15	1000	10	6.15 [4.68, 7.08]	5.24 [4.52, 5.93]	-0.91 [-1.42, -0.16]
R7 auto	1000	75	6.26 [2.93, 9.41]	6.58 [5.02, 8.49]	0.32 [-0.92, 2.09]
R8 auto	1000	75	7.73 [7.03, 8.90]	7.27 [3.05, 9.70]	-0.46 [-3.97, 1.79]
R8 CLAIM	1000	75	31.44 [20.52, 49.34]	33.22 [28.09, 35.80]	1.78 [-13.54, 11.32]
J2.10	1000	75	8.92 [8.46, 9.80]	7.99 [7.53, 8.37]	-0.92 [-2.27, -0.09]
J2.15	1000	75	5.52 [5.28, 5.71]	5.75 [4.52, 6.59]	0.23 [-0.76, 1.03]

The plain-read controls below report actual reference-to-survivor-loop-completion durations. Their nominal observation budget starts after the reference/kill bracket and equals twice the threshold. A complete outcome means that the message remained pending without redelivery through that procedure. These endpoints are available for the follow-up and are not reconstructed for the historical controls.

Trial	Redis	Δ, ms	Outcome	Observation, s
01/F0185	7.4.2	250	complete	0.509
01/F0186	7.4.2	250	complete	0.595
01/F0187	8.10.2	250	complete	0.699
01/F0188	8.10.2	250	complete	0.503
02/F0185	7.4.2	250	complete	0.507
02/F0186	7.4.2	250	complete	0.503
02/F0187	8.10.2	250	complete	0.503
02/F0188	8.10.2	250	complete	0.697
03/F0185	7.4.2	250	complete	0.511
03/F0186	7.4.2	250	complete	0.614
03/F0187	8.10.2	250	complete	0.511
03/F0188	8.10.2	250	complete	0.505

F.6 Compression-policy diagnostics

System	Payload	Inherited, kmsg/s	Off, kmsg/s	Off/inherited	$n_i/n_o/n_p$
Redis	Filler	5.05 [4.93, 5.18]	5.23 [5.12, 5.32]	1.04 [0.99, 1.06]	3/3/3
Redis	Template	5.16 [5.09, 5.25]	5.14 [5.12, 5.17]	1.00 [0.98, 1.02]	3/3/3
JetStream	Filler	0.67 [0.66, 0.68]	0.68 [0.67, 0.69]	1.02 [0.99, 1.04]	3/3/3
JetStream	Template	0.70 [0.66, 0.73]	0.69 [0.68, 0.70]	0.99 [0.94, 1.05]	3/3/3

Table 21: Current always-profile compression-policy sensitivity at sixteen publishers: equal-weight deployment means and ranges. Filler is the original compressible payload; Template is the fixed pseudorandom bank. Off/inherited is a matched throughput ratio; $n_i/n_o/n_p$ counts complete inherited/off trials/paired comparisons, each out of three. Both policies use the same Btrfs filesystem and NVMe device.

F.7 Failed follow-up outcomes

The table identifies each retained failed trial and its stage. Conf. and Unknown are client event classifications, not a broker-presence partition. Join is elapsed time from the timed phase's start until workers joined, when available. Stored is the count at a later inventory query. Inventory

queries are sequential, and timed-out operations can complete between them; these counts do not reconstruct the identities or confirmations of unknown calls. A dash marks an unavailable field, not zero. No failed observation is replayed into a planned success.

Trial	System	Mode	P	Stage	Conf.	Unknown	Join, s	Stored
01/F0006	J2.15.0	S	16	measurement	849	16	18.51	864
01/F0027	R8.10.2	S	1	measurement	531	1	12.36	532
01/F0054	J2.15.0	S	1	measurement	750	1	13.39	751
01/F0068	R7.4.2	S	16	warmup	–	–	–	–
02/F0019	J2.10.24	S	16	warmup	–	–	–	–
02/F0036	R7.4.2	S	16	warmup	–	–	–	–
02/F0045	R7.4.2	S	1	measurement	184	1	7.35	185
02/F0071	R8.10.2	S	1	warmup	–	–	–	–
03/F0046	J2.10.24	P	1	measurement	11208	1	7.62	11209
03/F0048	R7.4.2	S	1	measurement	299	1	8.15	300
03/F0063	R8.10.2	S	1	measurement	830	1	15.19	831

Trial	System	Profile	Unknown	Wait range, s	Error class
01/F0006	JetStream	Always	16	5.000096–5.001062	NATS timeout
01/F0027	Redis	Always	1	5.001448–5.001448	I/O timeout
01/F0054	JetStream	Always	1	5.000451–5.000451	NATS timeout
02/F0045	Redis	Always	1	5.001660–5.001660	I/O timeout
03/F0046	JetStream	Periodic	1	5.000530–5.000530	NATS timeout
03/F0048	Redis	Always	1	5.002365–5.002365	I/O timeout
03/F0063	Redis	Always	1	5.001360–5.001360	I/O timeout

Table 23: Recorded request waits in each measurement-stage failure. Ranges span unknown requests within the trial. Error classes are returned client errors; these times do not identify broker completion or storage-device latency. The NATS error text is `nats: timeout`; the Redis error text ends in `i/o timeout`.

F.8 Long requests in completed publication trials

The table lists all seven completed primary publication trials with a maximum confirmed-request latency of at least 1000 ms. All used always profiles. Four maxima exceed 2000 ms; the next is 1969.216 ms. The threshold is a post-observation descriptive selection for this table. Every listed trial remains in the complete primary summaries, and trials below the threshold remain in the full condition tables and machine-readable evidence. The publication admission budget is fifteen seconds; completion tails remain in the throughput denominator. These counts must not be combined with untimed warm-up failures to estimate a frequency of stalls in equal-duration windows.

Trial	System	Version	P	Maximum, ms
01/F0049	Redis	7.4.2	1	4182.712
03/F0036	Redis	8.10.2	1	3407.359
01/F0029	JetStream	2.10.24	16	3275.200
02/F0023	Redis	7.4.2	1	2296.818
03/F0039	JetStream	2.15.0	16	1969.216
02/F0034	JetStream	2.10.24	16	1749.164
03/F0050	JetStream	2.15.0	16	1312.415

Table 24: Largest confirmed-request latency in each completed primary publication trial meeting the descriptive 1000 ms threshold.

Complete paired contrasts, trial memberships and unavailable-pair counts are in `followup/derived/statistics.json`. Individual event, telemetry and syscall observations remain in the campaign directories and trial table. Section 5.6 presents synchronization diagnostics.

References

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013. URL: https://www.vldb.org/pvldb/vol6/p1033-akidau.pdf, doi:10.14778/2536222.2536229.

[2] Apache Kafka contributors. Apache Kafka 4.1 design: Message delivery semantics and using transactions. Official versioned documentation. Accessed 2026-09-24. URL: https://kafka.apache.org/41/design/design/.

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984. URL: https://www.cs.cornell.edu/courses/cs614/2004sp/papers/BN84.pdf, doi:10.1145/2080.357392.

[4] Btrfs contributors. Btrfs compression. Official filesystem documentation. Accessed 2026-09-24. URL: https://btrfs.readthedocs.io/en/latest/Compression.html.

[5] Btrfs contributors. Btrfs inode attributes. Official filesystem documentation. Accessed 2026-09-24. URL: https://btrfs.readthedocs.io/en/latest/ch-file-attributes.html.

[6] Btrfs contributors. btrfs-property(8): Inode properties. Official filesystem documentation. Accessed 2026-09-24. URL: https://btrfs.readthedocs.io/en/latest/btrfs-property.html.

[7] C. J. Clopper and E. S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934. doi:10.1093/biomet/26.4.404.

[8] Manuel Coenen, Christoph Wagner, Alexander Echler, and Sebastian Frischbier. Benchmarking financial data feed systems. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, DEBS '19, pages 252–253. ACM, 2019. Poster; author manuscript deposited in 2020. URL: https://arxiv.org/abs/2010.15534, doi:10.1145/3328905.3332506.

[9] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction: CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. URL: https://lean-lang.org/papers/lean4.pdf, doi:10.1007/978-3-030-79876-5_37.

[10] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus RabbitMQ. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 227–238. ACM, 2017. The inspected arXiv version is an extended author report following the conference paper. URL: https://arxiv.org/abs/1709.00333v1, doi:10.1145/3093742.3093908.

[11] HdrHistogram contributors. A high dynamic range (HDR) histogram. Software and coordinated-omission documentation. Accessed 2026-09-23. URL: https://github.com/HdrHistogram/HdrHistogram.

[12] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 63–74. ACM, 2013. URL: https://kar.kent.ac.uk/33611/, doi:10.1145/2464157.2464160.

[13] Kyle Kingsbury. NATS 2.12.1. Jepsen technical report, December 2025. Published 2025-12-08. Accessed 2026-09-23. URL: https://jepsen.io/analyses/nats-2.12.1.

[14] Jaeyeon Lim, Sewan Gu, and Dongweon Yoon. Performance analysis of event-driven autoscaling in Kubernetes environment. *Journal of Korean Institute of Information Technology*, 24(8):159–169, 2026. Full publisher HTML inspected; article in Korean. Accessed 2026-09-24. URL: https://ki-it.com/_common/do.php?a=full&aidx=50551&b=12&bidx=4529, doi:10.14801/jkiit.2026.24.8.159.

[15] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276. ACM, 2009. URL: https://research.ibm.com/publications/producing-wrong-data-without-doing-anything-obviously-wrong, doi:10.1145/1508244.1508275.

[16] OpenMessaging contributors. OpenMessaging Benchmark Framework, 2026. Software, revision 5b1fa70951a323da26bd587174b58bb2c65b0b5c. Accessed 2026-09-24. URL: https://github.com/openmessaging/benchmark/tree/5b1fa70951a323da26bd587174b58bb2c65b0b5c.

[17] Tapajit Chandra Paul, Pawissanutt Lertpongrujikorn, Hai Duc Nguyen, and Mohsen Amini Salehi. Benchmarking message brokers for IoT edge computing: A comprehensive performance study, 2026. Preprint, version 1. URL: https://arxiv.org/abs/2603.21600v1, arXiv:2603.21600, doi:10.48550/arXiv.2603.21600.

[18] Redis. Redis persistence. Accessed 2026-09-23. URL: https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/.

[19] Redis. XACK command documentation. Accessed 2026-09-23. URL: https://redis.io/docs/latest/commands/xack/.

[20] Redis. XAUTOCLAIM command documentation. Accessed 2026-09-23; version-specific statements cross-checked against Redis 7.4.2 source. URL: https://redis.io/docs/latest/commands/xautoclaim/.

[21] Redis. XREADGROUP command documentation. Accessed 2026-09-23; version-specific statements cross-checked against Redis 7.4.2 source. URL: https://redis.io/docs/latest/commands/xreadgroup/.

[22] Redis contributors. Redis 7.4.2 annotated configuration, 2025. Source code, release 7.4.2. Accessed 2026-09-23. URL: https://github.com/redis/redis/blob/a0a6f23d997b024689ba157916837f493a593a34/redis.conf.

[23] Redis contributors. Redis 7.4.2 append-only file implementation: src/aof.c, 2025. Source code, release 7.4.2. Accessed 2026-09-23. URL: https://github.com/redis/redis/blob/a0a6f23d997b024689ba157916837f493a593a34/src/aof.c.

[24] Redis contributors. Redis 7.4.2 release, 2025. Released 2025-01-06. Accessed 2026-09-23. URL: https://github.com/redis/redis/releases/tag/7.4.2.

[25] Redis contributors. Redis 7.4.2 stream implementation: src/t_stream.c, 2025. Source code, release 7.4.2. Accessed 2026-09-23. URL: https://github.com/redis/redis/blob/a0a6f23d997b024689ba157916837f493a593a34/src/t_stream.c.

[26] Redis contributors. Redis 8.10.2 append-only file implementation, 2026. Pinned source code. Accessed 2026-09-24. URL: https://github.com/redis/redis/blob/498ecd0d6d007db11ddb3aea9428552598a78622/src/aof.c.

[27] Redis contributors. Redis 8.10.2 event-loop and blocked-client scheduling, 2026. Pinned source: blocked.c, lines 825–852; ae.c, lines 382–403; server.c, lines 1572–1585 and 1855–1859; server.h, lines 119–123. Accessed 2026-09-24. URL: https://github.com/redis/redis/tree/498ecd0d6d007db11ddb3aea9428552598a78622/src.

[28] Redis contributors. Redis 8.10.2 release, 2026. Released 2026-09-17. Source commit 498ecd0d6d007db11ddb3aea9428552598a78622. Accessed 2026-09-24. URL: https://github.com/redis/redis/releases/tag/8.10.2.

[29] Redis contributors. Redis 8.10.2 stream implementation, 2026. Pinned source code. Accessed 2026-09-24. URL: https://github.com/redis/redis/blob/498ecd0d6d007db11ddb3aea9428552598a78622/src/t_stream.c.

[30] Redis Go client contributors. go-redis v9.7.0 command-specific socket timeout calculation. Pinned source code. Accessed 2026-09-24. URL: https://github.com/redis/go-redis/blob/ed37c33a9037483ad2a6b1042e5eb6df89009a1c/redis.go#L470-L478.

[31] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, 2009. URL: https://www.cl.cam.ac.uk/research/srg/opera/publications/papers/SamKai08-PerfEvalJ-SPECjms2007.pdf, doi:10.1016/j.peva.2009.01.003.

[32] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984. URL: https://web.mit.edu/saltzer/www/publications/endtoend/endtoend.pdf, doi:10.1145/357401.357402.

[33] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *3rd Symposium on Networked Systems Design and Implementation*, NSDI '06, pages 239–252. USENIX Association, 2006. URL: https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale.

[34] Gil Tene. wrk2: A constant throughput, correct latency recording variant of wrk. Software and measurement-method documentation. Accessed 2026-09-23. URL: https://github.com/giltene/wrk2.

[35] The Lean Developers. Lean 4.24.0 release notes, October 2025. Released 2025-10-14. Source tag: leanprover/lean4 v4.24.0. Accessed 2026-09-23. URL: https://lean-lang.org/doc/reference/latest/releases/v4.24.0/.

[36] The NATS Authors. JetStream consumers. Official documentation, revision f115becf6563e3bbe16bb94cbf87bfceb84199c1. Accessed 2026-09-23. URL: https://github.com/nats-io/nats.docs/blob/f115becf6563e3bbe16bb94cbf87bfceb84199c1/nats-concepts/jetstream/consumers.md.

[37] The NATS Authors. nats.go v1.39.1 JetStream fetch and acknowledgment implementation. Source code. Accessed 2026-09-23. URL: https://github.com/nats-io/nats.go/blob/4ba1afe8709b048ae89888f0f94da7eb8497e7ec/js.go.

[38] The NATS Authors. NATS Server v2.10.24 configuration parser: server/opts.go, 2024. Source code, release v2.10.24. Accessed 2026-09-23. URL: https://github.com/nats-io/nats-server/blob/1d6f7eaf0bd966dd02c95b8d5f624868f1a62544/server/opts.go.

[39] The NATS Authors. NATS Server v2.10.24 consumer implementation: server/consumer.go, 2024. Source code, release v2.10.24. Accessed 2026-09-23. URL: https://github.com/nats-io/nats-server/blob/1d6f7eaf0bd966dd02c95b8d5f624868f1a62544/server/consumer.go.

[40] The NATS Authors. NATS Server v2.10.24 file store: server/filestore.go, 2024. Source code, release v2.10.24. Accessed 2026-09-23. URL: https://github.com/nats-io/nats-server/blob/1d6f7eaf0bd966dd02c95b8d5f624868f1a62544/server/filestore.go.

[41] The NATS Authors. NATS Server v2.10.24 release, 2024. Released 2024-12-17. Accessed 2026-09-23. URL: https://github.com/nats-io/nats-server/releases/tag/v2.10.24.

[42] The NATS Authors. NATS Server v2.15.0 consumer implementation, 2026. Pinned source code. Accessed 2026-09-24. URL: https://github.com/nats-io/nats-server/blob/eb763679aa3c24a40dcd3012aa046ad1996d851c/server/consumer.go.

[43] The NATS Authors. NATS Server v2.15.0 file store, 2026. Pinned source code. Accessed 2026-09-24. URL: https://github.com/nats-io/nats-server/blob/eb763679aa3c24a40dcd3012aa046ad1996d851c/server/filestore.go.

[44] The NATS Authors. NATS Server v2.15.0 release, 2026. Released 2026-09-17. Source commit eb763679aa3c24a40dcd3012aa046ad1996d851c. Accessed 2026-09-24. URL: https://github.com/nats-io/nats-server/releases/tag/v2.15.0.