Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Cache Recovery Under Failure: Measuring the Trade-off Between Data Freshness and Database Load in Redis and Memcached

Kunal Dhanda

kunaliitkgp09@gmail.com

IIT Kharagpur: Indian Institute of Technology Kharagpur https://orcid.org/0009-0000-3113-6902

Research Article

Keywords: Redis, Memcached, cache consistency, cache recovery, stale reads, cache invalidation, PostgreSQL, fault tolerance.

Posted Date: November 3rd, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11587056/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Cache Recovery Under Failure: Measuring the Trade-off Between Data Freshness and Database Load in Redis and Memcached

Kunal Dhanda

November 2026

Abstract

In-memory caches such as Redis and Memcached sit in front of most relational databases, and teams choose between them largely on feature lists and raw throughput. We ask a narrower question that prior work leaves open: when the cache fails and comes back, how much stale data does it serve, and how much extra load does the database absorb? We built a testbed with PostgreSQL as the source of truth, five cache configurations (Memcached cold and warm restart; Redis without persistence, with AOF, and with replica failover), three invalidation protocols, and an oracle that checks every cache read against the database's commit history. Across 178 runs and 7.55 million reads, a 5 s cache outage under best-effort cache-aside invalidation produced a clear trade-off. Configurations that come back empty served essentially no persistent stale data but needed a median of 26.5 s to regain their hit ratio and sent 5,273 extra queries to the database. Configurations that come back warm recovered within the 1 s measurement resolution and added only 164 queries, but served 4.37–4.64% of post-recovery reads stale, across a median of 124 keys. The stale reads come from invalidations that were lost while the cache was unreachable and that the restored state then hid, so the difference follows the recovery mode, not the engine: Memcached's warm restart behaved like Redis AOF. Retrying failed invalidations after reconnecting cut stale reads by a factor of at least 43 with no statistically detectable change in tail latency; re-sending a median of 158 deletes per run across the warm configurations raised net extra database queries from 164 to 310. Two extensions sharpen the picture. A network partition left even Memcached's cold-restart configuration warm, and it then served 4.65% stale reads: what matters is whether the cache's state survives. When the application lost its in-memory state during the outage, in a simulated restart, the in-memory retry queue lost its pending invalidations (1.77% stale reads), while a transactional outbox, which records them in the database, delivered them (0.06%) at the price of two extra statements per write; these application-state results are preliminary. Cache recovery is therefore a consistency decision as well as an availability decision.

Keywords: Redis, Memcached, cache consistency, cache recovery, stale reads, cache invalidation, PostgreSQL, fault tolerance.

1 Introduction

A cache in front of a database is normally discussed as a performance device: it absorbs reads, cuts latency, and lowers cost [17, 15]. Its failure is discussed in the same terms. When a cache node is lost, every key it held becomes a miss, the database absorbs a surge, and recovery takes minutes [19]; surges of this kind are a recurring trigger of metastable outages [7, 12].

Cache failure is also a consistency problem, and that side is rarely measured. With the common cache-aside pattern, the application updates the database and then deletes the cached copy. If the cache is unreachable when the delete is sent, the invalidation is lost. A cache that comes back empty forgets the stale value along with everything else. A cache that comes back *warm*—from a persistence file, a warm-restart image, or a promoted replica—brings the stale value back and serves it until the key expires or is written again.

Consistency mechanisms for cache-augmented databases are well studied: race-free invalidation [9], leases [17, 10], and transactional caches [20, 2, 3]. Redis and Memcached have also been compared many times, but almost always as standalone stores without a database behind them [1, 25]. A literature search (about fifteen queries in the Consensus scholarly search engine, reviewed at abstract level) found no study that compares recovery modes of the two engines, in front of a database, under failure, while measuring stale reads directly; a systematic review would be needed to establish this more firmly.

This paper makes three contributions:

- **A staleness oracle** that classifies every cache read as fresh or stale against the database's commit history, validated against planted ground truth (§4.6).
- **A controlled measurement** of database load versus stale reads after a cache outage, across five recovery modes of two engines, three invalidation protocols, and crash, restart, partition and application-crash faults (§5).
- **A practical finding**: warm recovery reintroduces stale values when invalidations issued while the cache was unreachable are not recorded and replayed. The trade-off is set by whether the cache's state survives the outage, not by which engine it is. A retry queue removes most of the staleness while the application stays up; a transactional outbox also survives a loss of application state.

2 Background and Related Work

Cache-aside and its races. In cache-aside, reads check the cache and fall back to the database on a miss, then populate the cache; writes update the database and then delete the cached entry. Even without failures this pattern has races: a reader can fetch an old row, a writer can commit and invalidate, and the reader can then install the old row in the cache. Gumball [9] prevents such races in cache-augmented SQL systems, and the IQ framework uses Inhibit and Quarantine leases to give strong consistency without modifying the database [10]. Facebook's memcache deployment uses leases against stale sets and thundering herds [17]. Write-through and write-back policies update the cache instead of invalidating it, and a write-back design can preserve ACID transactions in a cache-augmented store [24]; we study the far more common invalidation-based cache-aside. TxCache [20] and monotonic consistent caching [2, 3] provide transactional views over memcached or Redis, using version information to decide whether a cached item may be served. These works treat the engine as interchangeable and provide guarantees on the read path. Our study is complementary: we measure how often an unprotected cache-aside deployment, the common case in practice, serves stale data after a cache outage, how that depends on the recovery mode, and how much a simple repair of lost invalidations recovers. A version check on the read path would also reject the stale values we observe, at the cost that Mao et al. measure [15].

Durability and failure. Redis offers append-only-file (AOF) persistence and asynchronous replication; Memcached offers neither by default, though recent versions can restore their memory image after a graceful restart, and research prototypes use persistent memory to avoid cold restarts [16, 13]. Redis replication can lose writes that were already acknowledged [8], and Redis Cluster loses data under majority node failure [22]. Cold caches after a reconfiguration can take about 18 minutes to recover [19]; load surges after cache loss can push systems into metastable states [7, 12], and request coalescing or probabilistic early expiration limit stampedes [23].

Measuring staleness. The BG benchmark treats stale ("unpredictable") data as a first-class metric for cache-augmented databases [6], and a later study compared client-server caches such as memcached and Redis with in-process caches using BG [11]. Mao et al. show that even a minimal consistency version check erases most of a cache's cost advantage [15]. Soni et al. measure stale-read rates of Cassandra and MongoDB under injected faults [21], and Jepsen-style history checking finds isolation anomalies in real databases [14]. None of these isolates the effect of a cache's recovery mode on staleness.

Reliable invalidation delivery. Delivering a change made in a database to another system without losing it is an established problem. The transactional outbox pattern writes an event into the same transaction as the data change and relays it afterwards [18]; change-data-capture systems such as LinkedIn's Databus and Netflix's DBLog read committed changes from the database log and feed caches, indexes and other derived stores [4, 5]. Our P3 protocol is a plain outbox; its novelty here is not the mechanism but the measurement of what it buys a cache that recovers warm. Table 1 summarises how this study relates to each line of work.

3 Research Questions

RQ1 Without faults, do Redis and Memcached give different stale-read rates under the same protocol? *H1: no; staleness is set by the protocol, not the engine.*
RQ2 After a cache outage, how do database load and stale reads compare across recovery modes? *H2: warm recovery gives less extra database load but more stale reads than cold recovery.*
RQ3 Does retrying failed invalidations remove the staleness, and at what cost? *H3: it removes most of it with little latency cost.*
RQ4 Does a cache that keeps its state through a connection-refusing network partition behave like a warm restart? *H4: yes; even a cold-restart engine serves stale data after a partition, because its state survives.*
RQ5 Does a durable invalidation outbox protect against a loss of application state, or a kill of the application process, during the outage, and at what cost? *H5: it does, where the in-memory retry queue loses the invalidations it held, at the price of slower writes.*
RQ6 How do write ratio and dataset size change the trade-off? We treat this question descriptively.
The proposal behind this study also asked when a cold restart pushes the database into a metastable state; that question needs a saturated database and is left to future work (§7).

4 Method

4.1 Testbed

All components ran on one Apple M2 MacBook Pro (8 cores, 16 GB RAM, macOS 26.5): PostgreSQL 14.17 as the source of truth, Redis 8.2.0, Memcached 1.6.45, and a Python 3.14 load generator using redis-py 8.1.0, pymemcache 4.0.0 and psycopg 3.3.6. Each cache server was started fresh for every run on a private port. Cache clients used a 50 ms connect timeout and a 200 ms socket timeout with no automatic retries, so a failed cache operation fell back to the database within milliseconds.

4.2 Cache configurations

Table 2 lists the five configurations. They span the three ways a cache can come back after an outage: empty (M1, R1), restored from local state (R2, M2), or replaced by a replica (R3). For R3 the harness promoted the replica with `REPLICAOF NO ONE` and redirected clients when the

Table 1: Prior work compared with this study, as each paper's own abstract or summary describes it; "–" means the aspect is not addressed in that description.

Work	Cache or store	Backing store	Faults	Consistency mechanism	Stale reads	DB load	App failure
TxCache [20]	Application cache	Relational DB	–	Transactional cache	–	yes	–
Gumball [9]	memcached	Relational DB	–	Race prevention	yes	–	–
IQ leases [10]	Key-value store	Relational DB	–	Inhibit/Quarantine leases	yes	–	–
BG [6]	memcached	Relational DB, document store	Named as a use case	(benchmark)	yes	yes	–
MCC [2, 3]	Redis, Memcached	HBase	–	Monotonic consistent caching	–	yes	–
Persistent Memcached [16, 13]	Memcached	–	Restart	Persistent memory	–	–	–
Read hints [19]	Replicated key-value store	–	Reconfiguration, failover	Warm replica caches	–	yes	–
Linearizable cache [22]	Redis Cluster, own cache	–	Node failure, partition, delay	Linearizability	yes	–	–
Cost of caches [15]	Distributed caches	Storage service	–	Version check	–	yes	–
Consistency knobs [21]	–	Cassandra, MongoDB	Several, incl. node kill	Tunable consistency	yes	–	–
This study	Redis, Memcached	PostgreSQL	Crash, restart, partition, app-state loss, process kill	Best effort, retry queue, outbox	yes	yes	yes

Table 2: Cache configurations.

ID	Engine and setting	State after the outage
M1	Memcached	Empty (cold)
R1	Redis, no persistence	Empty (cold)
R2	Redis, AOF `everysec`	Restored from AOF
M2	Memcached, `-e` memory file	Restored after graceful stop
R3	Redis + async replica	Replica promoted

outage ended, standing in for Redis Sentinel. R2 ran with `appendonly yes`, `appendfsync everysec` and RDB snapshots disabled (`save ""`); the AOF stayed far below the default 64 MB automatic-rewrite threshold, so no rewrite ran during a run, and after a crash Redis replayed the AOF as it stood on disk, which may lack up to about a second of writes. M2 used Memcached's restartable cache: item memory is mapped from a file given with `-e`, `SIGUSR1` stops the server after saving its metadata, and a restart with the same file and memory size restores the items. For every warm configuration we checked that the restored state was complete by the hit ratio, which returned to its pre-fault level within the first second after recovery (Table 5).

4.3 Invalidation protocols

P1, best effort. Cache-aside with delete-on-write; a delete that fails because the cache is unreachable is dropped. This is the default behaviour of most application code. **P2, retry queue.** As P1, but a failed delete is recorded in an in-memory map from key to a failure counter, and a background thread retries every recorded key until its delete succeeds; after a failed attempt it waits 50 ms. Repeated failures for one key coalesce into one entry, so the queue is bounded by the number of distinct keys written during the outage. Deletes are idempotent and carry no value, so replay order does not matter and a replay can at worst turn a fresh entry into a miss. Concurrency is handled with a per-key generation counter: a failed delete increments the key's counter under a lock; the replayer snapshots the pair (key, generation), sends the delete, and on success removes the entry only if the counter still holds the generation it snapshotted. A failure recorded while a replayed delete is in flight therefore raises the counter, the entry survives, and the key is retried, so an old successful delete can never clear a newer pending invalidation (Figure 1). Concurrent writers to one key coalesce into one entry; the replayer retries with a fixed 50 ms back-off; a duplicate delete can at worst turn a fresh entry into a miss; the queue holds at most one entry per distinct key written during the outage (at most 179 in our runs); and if the replayer's process fails, the queue is lost with it, which the process-kill fault tests. A unit test holds a replayed delete in flight, records a newer failure for the same key, releases the delete, and checks that the key is sent again rather than dropped; it passes.

```
on delete failure for key k:
    with lock: gen[k] = gen.get(k, 0) + 1
replayer, repeatedly:
    for (k, g) in snapshot(gen):
        if cache.delete(k) succeeds:
            with lock:
                if gen.get(k) == g: remove gen[k]
        else:
            sleep 50 ms; break
```

Figure 1: P2's retry queue. A failure recorded while a delete for the same key is in flight raises its generation, so the entry survives and is retried.

The queue is not durable: if the application process fails during the outage, its pending invalidations are lost. **P3, transactional outbox.** Each update inserts a row naming the key into an `outbox` table in the same database transaction as the update. After the commit the application deletes the key from the cache and, if that succeeds, deletes its outbox row. A dispatcher thread polls the outbox every 50 ms for rows older than 100 ms, deletes their keys from the cache, and removes the rows it delivered. Because the outbox commits with the data, no invalidation can be lost when the application fails. The cost is two extra statements per write (the insert and the delete) and a polling query per 50 ms. The outbox makes the delivery of each invalidation durable; it does not by itself prevent cache-aside fill races, in which a reader fetches an old row before a commit and installs it after that commit's delete, and we measure how often those still occur (§5.7). A second unit test exercises three failure windows of P3 against the real database and cache: a crash before the commit leaves neither the update nor an outbox row; a crash after the commit but before the cache delete is repaired by the dispatcher; and a crash after the delete but before the outbox row is removed leads to a second, harmless delete.

4.4 Faults and timeline

Each run lasted 50 s: 0–5 s warm-up, 5–15 s baseline, a fault at 15 s, recovery at 20 s, and observation until 50 s. A *crash* sent `SIGKILL` to the cache (or the Redis primary). A *restart* stopped it gracefully (`SIGTERM` for Redis, `SIGUSR1` for Memcached's warm restart). At recovery the cache was restarted from its own state or the replica was promoted. A separate set of runs varied the outage length (2, 5 and 10 s) for R2.

Two further faults test the extensions. In an *application-state-loss* fault, the Redis AOF cache (R2) is crashed as above, and 2.5 s into the outage the application loses its in-memory state: the P2 retry queue is emptied, as it would be if the process restarted, while the P3 outbox lives in the database and survives. This simulates the loss of volatile state, not a full process crash: database connections, in-flight transactions and the dispatcher are not interrupted. A *process kill* tests the full crash. The application runs as a separate process that appends every log record to disk as it is produced. The Redis AOF cache is crashed as above; 2.5 s into the outage the application process is killed with `SIGKILL` and a new one is started at once, continuing the request sequence; the cache recovers at 20 s. The kill loses everything the process held: the P2 queue, requests and transactions in flight, and its database and cache connections. PostgreSQL rolls back transactions that were not committed, so a P3 update and its outbox row commit or vanish together. Writes that committed but whose log record the kill lost are counted from the final database versions (a median of [pending] per P2 run and [pending] per P3 run); the oracle has no record of those versions and can only under-count around them. In a *partition*, the cache keeps running with all its data but becomes unreachable: clients connect through a TCP proxy that, at the fault, drops every open connection and stops accepting new ones, and at recovery accepts connections again. Clients see refused connections, as they would with a rejecting firewall rule, rather than timeouts.

The two fault types are not interchangeable, so each comparison uses one of them throughout. Memcached's warm restart restores its image only after a graceful stop; after `SIGKILL` it comes back empty and is identical to M1, so M2 is run only with a graceful restart. The engine comparison between warm configurations (M2 against R2) therefore uses a graceful restart for both, and R2 is also run under a crash so that the effect of the fault type can be seen for one configuration.

4.5 Workload

The database held 10,000 rows, each with a version number and a 100-byte value. Eight client threads issued a paced 1,000 requests/s in total, 95% reads and 5% writes, over keys drawn from a Zipf distribution with exponent 0.99; the same keys were hot in every run. Cached values carried the row version and a 300 s TTL, longer than a run, so expiry could not hide staleness. The cache was preloaded before each run. For RQ6, crash runs of M1 and R2 were repeated with 1% and 20% writes and with a 100,000-row dataset.

4.6 Staleness oracle

Writers record the commit time t_c of each new version immediately after the `UPDATE` returns; readers record the start time t_s of each read and the version v it returned. A read of key k is *stale* if

$$\max\{\, v' : (k, v', t_c) \in W,\ t_c < t_s \,\} > v, \qquad (1)$$

where W is the set of committed writes.

Worked example. Let key 42 hold version 7 in both the cache and the database. A writer commits version 8 and records $t_c = 100.4$ ms; its delete is acknowledged at 100.9 ms. A read that starts at 100.6 ms and gets version 7 from the cache is stale with age 0.2 ms: invalidation lag, below the 10 ms threshold. If the cache had been unreachable and the delete had failed, a read at 2,300 ms returning version 7 would be stale with age 2,199.6 ms and would count as persistent. A read that starts at 100.2 ms, before t_c, and returns version 7 is not counted at all, even if the commit had in fact already happened; that is the conservative choice that makes the oracle under-count rather than over-count.

Ordering and concurrency. Versions are assigned by the database itself (`SET version = version + 1`) under PostgreSQL's row lock, so concurrent writers to the same key are serialized and each key's versions are unique and increase in commit order. Every version is committed before its t_c is taken, so a read that starts after t_c and returns an older version is stale by construction: the oracle can miss staleness but cannot invent it. It misses stale reads that start between the true commit and t_c, a window bounded by the time from sending the `UPDATE` to its acknowledgement (median 1.11 ms, 99th

percentile 9.24 ms). Cache repopulation races and invalidations delivered late or out of order need no special handling: whatever path installs an old version in the cache, a later read of it is compared against the commit history.

Staleness age and threshold. The *age* of a stale read is t_s minus the commit time of the first version newer than v. Very young stale reads are *invalidation lag*: the time between a commit and the acknowledgement of its delete, which across 462,870 successful invalidations had a median of 0.41 ms and a 99th percentile of 3.77 ms. We therefore report *persistent* stale reads, older than 10 ms, throughout; §5.9 shows that the results do not depend on this choice between 1 ms and 1 s.

We validated the oracle with two controls. As a negative control, reads served by the database must never be stale. As a positive control, 6 runs planted 50 lost invalidations by updating hot rows without touching the cache; the oracle must flag hits on exactly those keys until each is next invalidated.

4.7 Statistics

Of 306 planned runs, 178 completed validly, all in a shuffled order. The main cells (no fault, crash and restart under P1 and P2) ran 5–10 times each, the principal extension comparisons (P2 against P3 after an application-state loss, and P3 without faults) 2–5 times, and the other extension and sensitivity cells 1–5 times. 2 runs during which the laptop went to sleep were discarded and rerun, and 1 run whose preload timed out was retried; the manifest `results/manifest.csv` lists every run with its seed, completion time, duration and status. Run r of every cell used the same seed, and so the same request sequence, which pairs runs across configurations and explains why some cells share run-level values. The experimental run, not the request, is the unit of analysis: all statistics are computed over per-run values, which Tables 10 and 11 list in full. We report per-cell medians with 95% percentile-bootstrap confidence intervals from 10,000 resamples of the run-level values; with so few runs such an interval can only take observed values as its bounds, so it is coarse. Hypotheses are tested with two-sided Mann–Whitney U tests and Cliff's δ as the effect size, Holm-corrected within two families: the 9 tests of the main hypotheses (H1–H3) and the 7 tests of the extensions (H4, H5). Hypotheses about recovery mode are tested on runs pooled by mode: warm against cold (19 and 17 runs) for H2, and P2 against P1 over the four warm cells (32 and 32 runs) for H3. Because run r of every cell shares its request sequence, each test is repeated as a paired Wilcoxon signed-rank test on the differences between matched runs, Holm-corrected in the same families; Table 9 reports both. Confidence intervals are not reported for cells with fewer than three runs. Table 9 lists every test, and Table 3 counts every run from the logs.

Table 3: Experiment inventory, counted from the run logs.

Experiment	Cells	Runs per cell	Runs
Baseline, no fault	3	2–9	16
Crash, 5 s	9	2–10	71
Graceful restart	5	2–8	30
Application-state loss	2	4–5	9
Partition	3	3–5	11
Outage length (2, 10 s)	2	5	10
Write ratio (1%, 20%)	6	1–5	21
Dataset size (100k rows)	2	2	4
Oracle control (planted)	2	3	6
Total	34		178

Table 4: Research questions and the evidence that answers them.

RQ	Claim tested	Evidence	Tests
RQ1	H1: engines alike without faults	§5.2, Table 5	H1
RQ2	H2: warm recovery trades load for staleness	§5.3, Figs. 2–3, Table 5	H2 (3)
–	Engine beyond recovery mode	§5.4, equivalence check	Eng (2)
RQ3	H3: retry queue removes staleness	§5.5, Fig. 4, Table 5	H3 (3)
RQ4	H4: a partition leaves the cache warm	§5.6, Table 6	H4 (2)
RQ5	H5: outbox survives app-state loss and a process kill; its cost	§5.7, Fig. 4, Table 6	H5 (7)
RQ6	Write ratio and dataset size	§5.8, Table 7	descriptive

5 Results

The load generator sustained 996–1001 operations/s in every run, including during outages. Table 4 maps each research question to the evidence and tests that answer it.

5.1 The oracle is exact on planted ground truth

Across all 178 runs, 0 database-served reads were flagged stale. In the 6 positive-control runs we rebuilt the expected stale reads independently from the planted writes: every hit on a planted key between the plant and that key's next successful delete. The oracle flagged 3,010 of the 3,010 expected reads, missed 0, and flagged 0 other reads on planted keys. It also flagged 164 persistent stale reads on keys that were not planted. All 164 of them trace to a cache-aside fill race: a reader fetched the row just before a write and installed it in the cache just after the write's delete. These are real stale reads, not oracle errors.

5.2 Without faults, the engines do not differ (RQ1)

With no faults, persistent stale reads were rare for both engines: a median of 0.00% for Memcached and 0.00% for Redis, and at most 0.07% in any run ($p = 0.551$, $\delta = +0.11$). The

few persistent stale reads that did occur came from the same fill races as above; invalidation lag under 10 ms accounted for the rest. No statistically detectable difference between the engines was observed without faults, and the exploratory equivalence check found the difference in medians within the margin; because the margin was chosen after seeing the data, this is consistent with H1 but does not establish general equivalence.

5.3 After a crash, recovery mode sets the trade-off (RQ2)

Figure 2 and Table 5 show the outcome of a 5 s crash. During the outage every configuration behaves the same: the cache is unreachable, all reads go to the database, and its query rate rises from about 91 to 998 queries/s, the full request rate: a median 11.0-fold amplification of database demand. The configurations diverge when the cache returns.

Cold recovery costs load. Memcached (M1) and Redis without persistence (R1) come back empty. Their hit ratio took a median of 26.5 s and 26.6 s to return to 90% of baseline, and the database absorbed 5,273 and 5,288 extra queries in the 30 s after recovery. They served essentially no stale data (0.00% and 0.00% of reads).

Warm recovery costs consistency. Redis with AOF (R2) and Redis with a promoted replica (R3) were back at their baseline hit ratio within the 1 s resolution of our measure, with only 176 and 122 extra database queries, close to the 115–129 that runs without any fault show from load drift. But each failed to apply the invalidations sent during the outage (median 245 failed deletes per run), so they restored the old values of the keys written in that window. A median of 4.63% (R2) and 4.64% (R3) of post-recovery reads were stale, spread over 124 and 125 keys. Pooled against the two cold configurations, warm recovery served more stale reads ($p_{\text{Holm}} < 0.001$, $\delta = +1.00$), caused fewer extra database queries ($p_{\text{Holm}} < 0.001$, $\delta = -1.00$) and regained its hit ratio sooner ($p_{\text{Holm}} < 0.001$, $\delta = -1.00$). Stale reads persisted until the key was next written: the median stale read was 8.1 s old for R2, and the oldest 34.2 s, bounded only by the end of the run.

Figure 3 plots every run in the two dimensions. Under best-effort invalidation the runs fall into two groups with no overlap: cold recovery in the lower right, warm recovery in the upper left. H2 holds: every Holm-adjusted test, unpaired and paired, is below 0.05. Runs with the retry queue sit in the lower left, the only region with both low database load and few stale reads. Among the crash cells, the medians on the Pareto frontier, where no other cell has both fewer stale reads and fewer extra queries, are M1/P1, R2/P1, R3/P1, R3/P2; no other cell is better on both counts.

5.4 Recovery mode, not engine

If the effect came from Redis itself, Memcached should not show it. It does. After a graceful restart, Memcached with a warm-restart memory file (M2) served 4.37% stale reads over 114 keys, against 4.52% over 122 keys for Redis with AOF ($p = 0.731$, $\delta = -0.14$). Both recovered their hit ratio within 1.0 s. Conversely, Redis without persistence (R1) behaved like cold Memcached after a crash ($p = 0.198$; Table 5). A non-significant test does not show that the engines behave the same, so we also checked equivalence: the 90% bootstrap interval of the difference in median stale reads (equivalent to two one-sided tests at 5%) against a margin of ± 1 percentage point, under a quarter of the cold–warm gap. For M2 against R2 the interval was [-0.68, +0.58] points, within the margin; for R1 against M1 it was [+0.00, +0.00], within it. The margin was chosen after seeing the data, so this is a descriptive check rather than a pre-registered test. The fault type mattered little for the one configuration run under both: Redis with AOF served 4.63% stale reads after a crash and 4.52% after a graceful restart. Under the tested workload and configurations, we found no difference in staleness attributable to the engine beyond the recovery mode it was configured with.

5.5 Retrying failed invalidations removes most staleness (RQ3)

With the retry queue (P2), the replayer re-sent a median of 160 deletes for R2 after a crash (158 pooled over the four warm cells) once the cache was reachable again. Persistent stale reads after a crash fell from 4.63% to 0.10% for R2 and from 4.64% to 0.09% for R3; after a restart, from 4.37% to 0.05% for M2 and from 4.52% to 0.10% for R2 (Figure 4). Pooled over the four warm cells, the reduction is significant ($p_{\text{Holm}} < 0.001$, $\delta = -1.00$). The remaining stale reads hit keys read in the moments between the cache coming back and their delete being replayed; the last replay finished a median of 0.15 s after recovery for R2.

The retry queue has a cost, but a small one. Each replayed delete turns a stale entry into a miss, so net extra database queries rose from a median of 164 to 310 ($p_{\text{Holm}} < 0.001$), still far below the 5,273 of cold recovery. Median post-recovery p99 latency was 6.52 ms without and 6.22 ms with the retry queue ($p_{\text{Holm}} = 1.000$), H3 holds: stale reads fell and database load rose significantly, while no difference in p99 latency was detected.

5.6 A connection-refusing partition leaves the cache warm (RQ4)

A partition leaves the cache process and its data intact. After a 5 s connection-refusing partition, Memcached (M1), which comes back empty after a crash, served 4.65% stale reads over 125 keys, against 0.00% after a crash ($p_{\text{Holm}} = 0.012$, $\delta = +1.00$). Redis without persistence (R1) behaved the same way: 4.67% after a partition against 0.00% after a crash ($p_{\text{Holm}} = 0.017$). Both regained their hit ratio within 1.0 s and cost the database only 119 and 112 extra queries: the warm-recovery profile, not the cold one. The retry queue removed this staleness too (0.06% with P2). These observations are consistent with H4: both cold-restart engines kept their contents through the partition and served more stale reads than

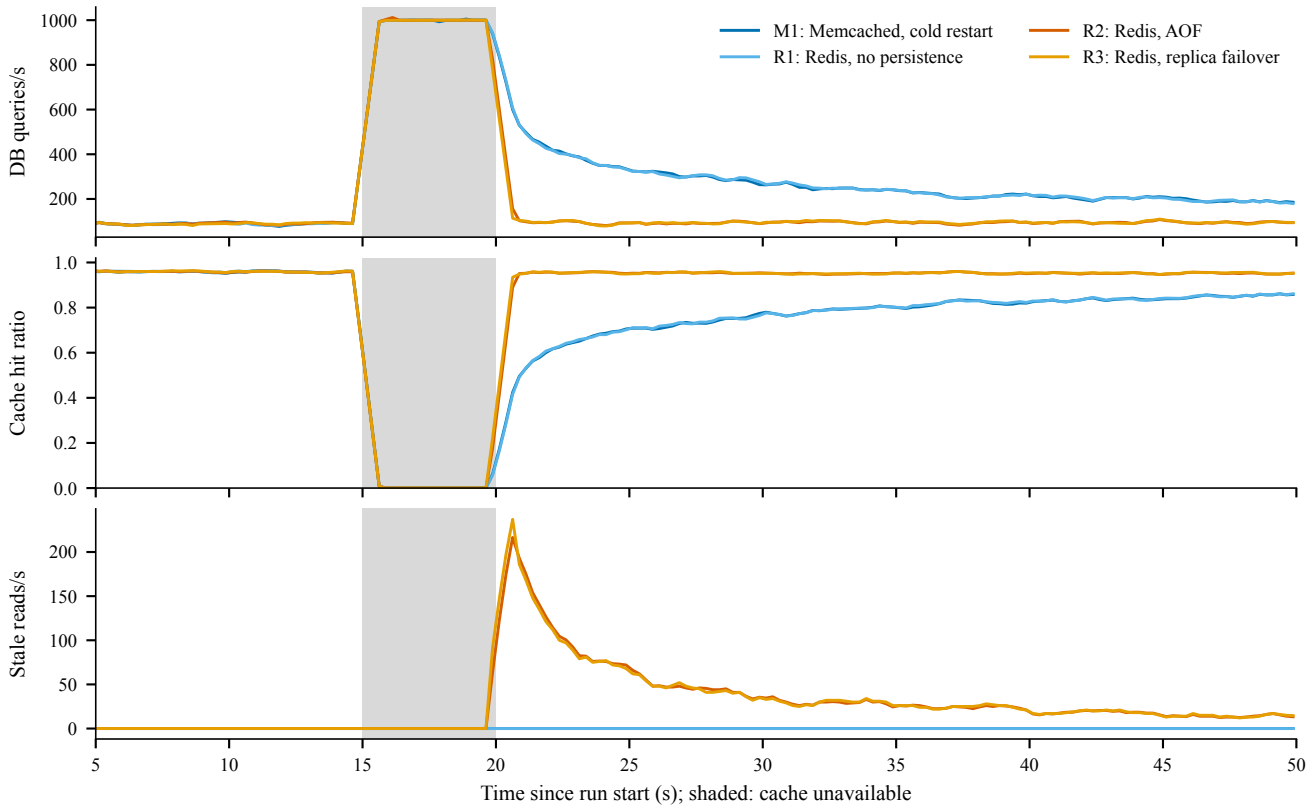


Figure 2: A 5 s cache crash under best-effort invalidation (P1); medians of 8–10 runs per configuration, 1 s smoothing. Cold configurations (M1, R1) send all traffic to the database until the cache refills; warm ones (R2, R3) recover at once but serve stale reads for keys written during the outage.

after a crash. With 5 and 3 runs, however, the Holm-adjusted tests do not establish it. Table 6 gives the full results.

5.7 The outbox survives a loss of application state; the retry queue does not (RQ5)

When the application lost its in-memory state 2.5 s into the outage, the P2 retry queue lost a median of 85 pending keys. Those invalidations were never delivered (median 68 undelivered keys per run), and 1.77% of post-recovery reads were stale. The P3 outbox kept its pending invalidations in the database; its dispatcher delivered them within 0.22 s of recovery, leaving 0 undelivered keys and 0.06% stale reads (4 and 5 runs; stale reads $p_{\text{Holm}} = 0.063$, $\delta = -1.00$; undelivered keys $p_{\text{Holm}} = 0.054$). In these runs the outbox delivered every pending invalidation and the retry queue did not, but with 4 and 5 runs the difference is not yet statistically established. Without the state loss, P3 and P2 performed alike after a cache crash (0.13% against 0.10%; $p_{\text{Holm}} = 0.533$).

With a real kill of the application process, the P2 retry queue left a median of [pending] keys undelivered and [pending]% of post-recovery reads stale, against [pending] keys and [pending]% with the outbox ([pending] and [pending] runs; stale reads $p_{\text{Holm}}[pending]$, $\delta = [pending]$; undelivered keys $p_{\text{Holm}}[pending]$). [pending] The two protocols therefore protect different failure boundaries: P2 covers a cache that becomes unavailable while the application stays up; P3 also covers an application that fails while invalidations are pending.

The outbox's cost is on the write path. Without faults, the latency of a write (update, cache delete and outbox bookkeeping) had medians over runs of 1.46, 3.67 and 10.66 ms at p50, p95 and p99 with P1, against 2.76, 7.27 and 19.70 ms with P3 (5 and 2 runs; p99 difference $p_{\text{Holm}} = 0.286$: no statistically detectable difference); each P3 write issued two extra statements, and the dispatcher queried the outbox 18 times per second. The outbox did not remove fill races: without faults, P3 runs had a median of 4 race-induced stale reads per run against 0 with P1 ($p_{\text{Holm}} = 0.412$). Fill races are a separate failure mode that leases address [17, 10]; durable delivery alone does not.

5.8 Write ratio and dataset size (RQ6)

Table 7 repeats the crash runs with 1% and 20% writes and with ten times as many rows. More writes during the outage meant more failed deletes (a median of 52, 245 and 1024 for R2 at 1, 5 and 20% writes) and more keys served stale after warm recovery (38, 124 and 296). Cold Memcached served 0.00% and 0.05% stale reads at 1 and 20% writes, and the

Table 5: Results after a 5 s outage. Medians over n runs; stale reads with 95% bootstrap CI. Stale reads and keys count persistent staleness (>10 ms) after recovery; hit recovery is the time until the 1 s hit ratio returns to 90% of baseline (resolution 1 s); extra DB queries are the queries in the 30 s after recovery minus the pre-fault rate over the same time. Negative values occur when the post-recovery rate falls below the pre-fault baseline; they are kept as measured, and medians are taken over the raw values.

Fault	Config	Prot.	n	Stale reads (%) [95% CI]	Stale keys	Hit recovery (s)	Extra DB queries	p99 latency (ms)
Crash	M1	P1	9	0.00 [0.00, 0.00]	0	26.5	5,273	6.75
Crash	M1	P2	8	0.00 [0.00, 0.07]	0	26.8	5,282	6.75
Crash	R1	P1	8	0.00 [0.00, 0.00]	0	26.6	5,288	7.15
Crash	R1	P2	8	0.00 [0.00, 0.04]	0	27.0	5,290	7.44
Crash	R2	P1	10	4.63 [3.87, 4.75]	124	1.0	176	6.89
Crash	R2	P2	8	0.10 [0.08, 0.17]	18	1.2	320	6.10
Crash	R2	P3	2	0.13 [–]	20	1.2	321	6.20
Crash	R3	P1	9	4.64 [3.88, 4.86]	125	1.0	122	7.07
Crash	R3	P2	9	0.09 [0.05, 0.23]	13	1.0	238	6.32
Restart	M2	P1	7	4.37 [3.84, 4.55]	114	1.0	186	5.16
Restart	M2	P2	8	0.05 [0.04, 0.07]	8	1.2	329	6.28
Restart	M2	P3	2	0.07 [–]	10	1.2	315	5.25
Restart	R2	P1	6	4.52 [3.80, 4.76]	122	1.0	159	7.07
Restart	R2	P2	7	0.10 [0.09, 0.14]	16	1.2	310	6.20

Table 6: Partition and application-failure results, 5 s outage. Medians over n runs; stale reads with 95% bootstrap CI. Failed keys: distinct keys whose invalidation failed while the cache was away. Undelivered: those the protocol never delivered afterwards.

Fault	Config	Prot.	n	Stale reads (%) [95% CI]	Stale keys	Failed keys	Undelivered	Replayed deletes	Extra DB queries
Partition	M1	P1	5	4.65 [3.49, 4.75]	125	156	156	0	119
Partition	R1	P1	3	4.67 [3.49, 4.69]	124	155	155	0	112
Partition	M1	P2	3	0.06 [0.05, 0.08]	12	148	0	148	231
App state loss	R2	P2	4	1.77 [1.53, 2.32]	68	154	68	86	250
App state loss	R2	P3	5	0.06 [0.05, 0.14]	12	158	0	256	329

retry queue cut the warm cache's stale reads to 0.05% and 0.13%. With 100,000 rows, cold recovery was much slower: in 2 of 2 runs the hit ratio of M1 had not regained 90% of its baseline by the end of the 30 s window, and the database absorbed 10,148 extra queries, against 5,273 with 10,000 rows. The warm cache served 3.87% stale reads over 106 keys. The direction of the trade-off held in every workload we tried; its size depends on how many keys are written during the outage and how many the cache must refill.

5.9 The results do not depend on the staleness threshold

Table 8 repeats the stale-read measurement with thresholds from 1 ms to 1 s. For Redis with AOF after a crash, the median share of stale reads was 4.63% above 1 ms, 4.63% above 10 ms, 4.63% above 100 ms and 4.59% above 1 s. Stale values restored by warm recovery stay stale for seconds, so almost all of them clear every threshold, and the cold configurations stay near zero at all four.

5.10 Staleness grows with outage length

For R2 after a crash, the number of keys served stale grew with the outage: a median of 57, 124 and 217 keys for 2, 5 and 10 s outages (Spearman $\rho = 0.92$, $p < 0.001$; Figure 5). Each write during the outage is a lost invalidation, so the exposure scales with write rate times outage length, while Zipf skew means the hottest keys are rewritten, and so repaired, soonest.

6 Discussion

What to do with this. Durability features are usually framed as pure gains for a cache: fewer misses, less database load, faster recovery. Our results show they are a trade under a specific condition: when invalidations issued while the cache is unreachable are dropped, a cache that comes back warm restores the values they were meant to remove, and under cache-aside nothing repairs those values until the key is written again or expires. Teams that enable AOF, warm restart or replica failover for a cache should pair it with one of four measures: replay failed invalidations on reconnect (our P2); write invalidation events to a transactional outbox in the same database transaction as the update, with a dispatcher that deletes the keys once the cache is reachable, which also survives a failure of the application; invalidate from the database's change log so that no invalidation depends on the application at all [5]; or treat any recovery as a cold start for keys written during the outage, for example by bumping a key-space version. Short TTLs bound the exposure but also lower the hit ratio.

Table 7: Sensitivity to write ratio and dataset size: crash, 5 s outage. Medians over n runs. Rows with 5% writes and 10k keys are the main cells.

Writes	Keys	Config	Prot.	n	Stale (%)	Stale keys	Failed deletes	Hit recovery (s)	Extra DB queries
1%	10k	M1	P1	5	0.00	0	40	25.5	5,529
1%	10k	R2	P1	1	5.29	38	52	1.0	104
1%	10k	R2	P2	4	0.05	6	38	1.0	151
5%	10k	M1	P1	9	0.00	0	249	26.5	5,273
5%	10k	R2	P1	10	4.63	124	245	1.0	176
5%	10k	R2	P2	8	0.10	18	252	1.2	320
20%	10k	M1	P1	3	0.05	2	992	25.5	4,253
20%	10k	R2	P1	4	4.07	296	1024	1.0	592
20%	10k	R2	P2	4	0.13	23	1030	1.6	722
5%	100k	M1	P1	2	0.00	0	245	–	10,148
5%	100k	R2	P1	2	3.87	106	240	1.2	286

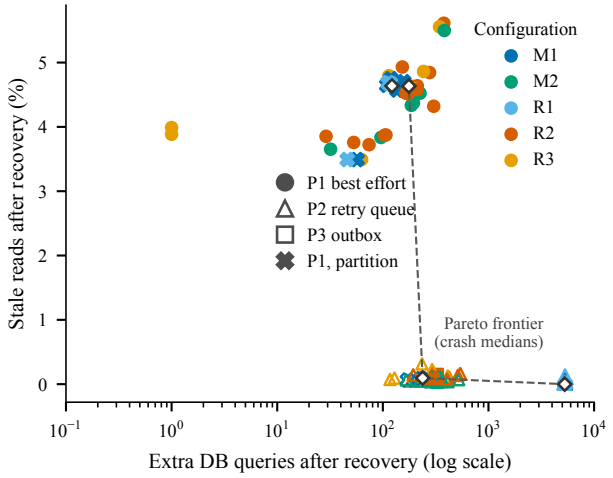


Figure 3: Each run after a 5 s outage. Cold recovery (M1, R1) trades database load for consistency; warm recovery (R2, R3, M2) and a connection-refusing partition do the reverse unless failed invalidations are delivered (P2, P3). Values below 1 are drawn at 1 on the log axis.

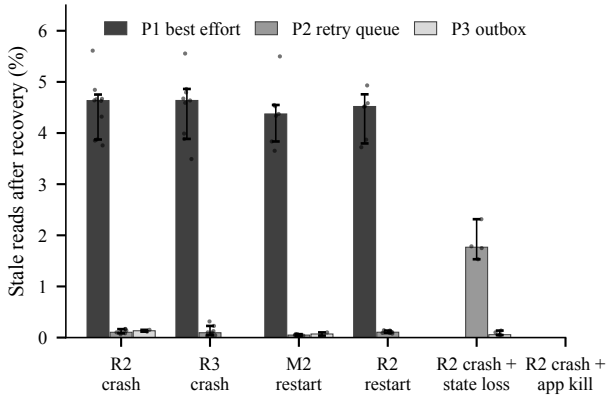


Figure 4: Stale reads after recovery for the warm configurations under each protocol: bars are medians with 95% bootstrap CIs, dots are individual runs. The two rightmost groups add, 2.5 s into the outage, a loss of application state and a real kill of the application process.

Two failure boundaries. The retry queue and the outbox are not alternatives so much as answers to different failures. P2 carries invalidations across a cache outage as long as the application process survives; P3 also carries them across a failure of the application, because the pending invalidations are committed with the data. Neither stops fill races, which come from the ordering of a read and a write rather than from a failure; leases or versioned cache writes are needed for those.

Choosing an engine. For this question the engine appears to be the wrong unit of choice: in our runs, Memcached with warm restart was as exposed as Redis with AOF, and Redis without persistence as safe as plain Memcached after a crash. The useful question is whether the cache will come back with state, and if so, how invalidations sent during its absence are delivered.

Relation to prior consistency work. Leases and transactional caches [10, 20, 3] close race windows in normal operation; they do not by themselves deliver an invalidation to a cache that was unreachable when it was sent. Our results suggest failure handling deserves the same attention as race handling: in our runs, failure produced far more stale reads than races did.

7 Threats to Validity

Scale and hardware. All components shared one laptop and talked over loopback, at 1,000 requests/s over 10,000 keys. The database was never saturated, so we measured extra database load, not overload, and could not study metastability. **Replication.** On loopback, Redis replication lag is near zero, so R3 almost never lost an acknowledged write before the crash; across a network, lost replication would add staleness that our numbers do not include. **Failover.** Replica promo-

Table 8: Median stale reads after recovery by staleness threshold, 5 s outage: percentage of reads, with the median number of stale reads per run in parentheses. The 10 ms column is the one used elsewhere.

Fault	Config	Prot.	>1 ms	>10 ms	>100 ms	>1 s
Crash	M1	P1	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
Crash	M1	P2	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
Crash	R1	P1	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
Crash	R1	P2	0.00 (0)	0.00 (0)	0.00 (0)	0.00 (0)
Crash	R2	P1	4.63 (1,323)	4.63 (1,323)	4.63 (1,320)	4.59 (1,308)
Crash	R2	P2	0.10 (30)	0.10 (30)	0.10 (29)	0.09 (25)
Crash	R3	P1	4.64 (1,318)	4.64 (1,318)	4.64 (1,318)	4.61 (1,311)
Crash	R3	P2	0.09 (27)	0.09 (27)	0.09 (25)	0.07 (21)
Restart	M2	P1	4.37 (1,247)	4.37 (1,247)	4.37 (1,247)	4.36 (1,243)
Restart	M2	P2	0.05 (14)	0.05 (14)	0.05 (14)	0.04 (12)
Restart	R2	P1	4.52 (1,287)	4.52 (1,286)	4.50 (1,281)	4.48 (1,274)
Restart	R2	P2	0.10 (30)	0.10 (30)	0.09 (26)	0.08 (23)

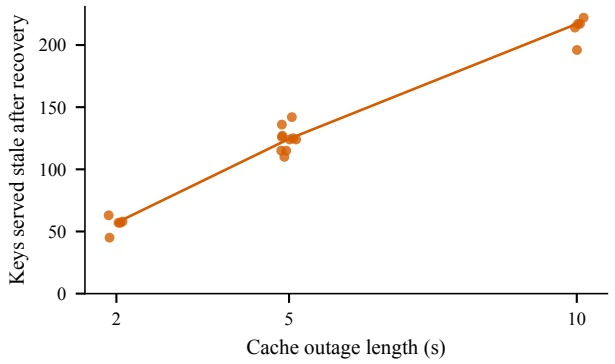


Figure 5: Keys served stale after recovery versus outage length (R2, crash, P1); one point per run, line through medians.

tion was done by the harness at a fixed 5 s, not by Sentinel's failure detection and election. **Client.** The load generator was a Python process with eight threads; absolute latencies reflect it and should not be compared with production systems, although all configurations shared it. **Workload.** The workload was synthetic: one table of 10,000 rows (100,000 in one sensitivity set), single-row reads and writes, write ratios of 1, 5 and 20%, one request rate, fixed Zipf skew and a 300 s TTL. Request rate, skew and TTL were not varied. **Faults.** We injected process crashes, graceful restarts, application restarts and connection-refusing partitions through a proxy, which adds a hop to every request in the partition runs. Partitions that drop packets, so that clients time out, and replication lag were not tested. The application-state-loss fault discards only volatile state. The process-kill fault kills the application at one moment, 2.5 s into the outage, and restarts it at once; kills at other moments, such as between a commit and its cache delete while the cache is up, were not targeted separately, and the restart is faster than a real deployment's would be. **Repetitions.** Ten runs per main cell and five per extension cell give limited power for per-cell comparisons; the main hypotheses were tested on pooled runs, and a failure to find a difference between engines is not evidence of equality. **Oracle.** Commit times are taken after the commit returns, so the oracle can under-count staleness; the planted-control runs show it does not over-count. **Run hygiene.** Two runs during which the laptop went to sleep were discarded and rerun: cache TTLs run on wall-clock time, so a sleep expires entries while the monotonic clock is paused. Later runs check the drift between the two clocks and reject themselves if it exceeds 2 s. Some runs also showed client stalls of a few hundred milliseconds; these affect latency and peak-rate figures but not the oracle, which works on timestamps.

8 Conclusion

A cache that survives its own failure with its data intact is not automatically better. Across 178 controlled runs, warm recovery—whether Redis AOF, a promoted Redis replica, or Memcached's warm restart—saved the database thousands of queries but, under best-effort invalidation, reintroduced stale values for the keys written while the cache was away; cold recovery did the reverse. The engine did not decide the outcome; the recovery mode and the invalidation protocol did. Retrying failed invalidations kept the benefits of warm recovery and removed most of its cost; a transactional outbox did the same and, in preliminary runs, also survived a loss of application state. A network partition, which preserves the cache's state, behaved like a warm restart even for an engine configured to restart cold. None of the protocols removed cache-aside fill races. Future work should repeat the measurement across separate hosts, with partitions that drop packets and with replication lag; at saturating load, to test whether cold restarts push a loaded database into a metastable state; with other request rates, skews and TTLs; and with change-log invalidation and leases.

Artifacts. The harness, oracle, raw read and write logs for every run, and the scripts that produce every number, table and figure in this paper are in the accompanying project folder (`cache-staleness-study`); its README lists software versions and the commands that reproduce the full matrix, the analysis and this PDF. All numbers in the text are generated from the run logs, not typed by hand.

References

[1] D. Almeida et al. Performance comparison of Redis, Memcached, MySQL, and PostgreSQL: a study on key-value and relational databases. In *SmartTechCon*, 2023. doi:10.1109/smarttechcon57526.2023.10391649.

[2] S. An and Y. Cao. Making cache monotonic and consistent. *PVLDB*, 16(4), 2022. doi:10.14778/3574245.3574271.

[3] S. An et al. Caching with monotonicity and consistency. *ACM Trans. Database Syst.*, 2026. doi:10.1145/3845609.

[4] S. Das et al. All aboard the Databus!: LinkedIn's scalable consistent change data capture platform. In *SoCC*, 2012. doi:10.1145/2391229.2391247.

[5] A. Andreakis et al. DBLog: a watermark based change-data-capture framework. arXiv:2010.12597, 2020.

Table 9: Hypothesis tests: two-sided Mann–Whitney U, Holm-corrected within each family (main hypotheses above the rule, extensions below); n gives the runs in each group. Under a crash with P1, *warm* pools R2 and R3 and *cold* pools M1 and R1; for H3, *warm* pools R2 and R3 after a crash and M2 and R2 after a restart. A positive Cliff's δ means the first group is larger.

Hyp.	Comparison	n	p	p_{Holm}	Pairs	Paired p_{Holm}	Paired median diff. [95% CI]	δ
H1	stale reads, no fault: M1 vs R1	9/5	0.551	1.000	5	1.000	+0 [+0, +0.07]	+0.11
H2	stale reads, crash: warm/cold	19/17	<0.001	<0.001	16	<0.001	+4.65 [+4.48, +4.79]	+1.00
H2	DB load, crash: warm/cold	19/17	<0.001	<0.001	16	<0.001	-5.1e+03 [-5.13e+03, -5.06e+03]	-1.00
H2	hit recovery, crash: warm/cold	19/17	<0.001	<0.001	16	0.002	-25.6 [-26.1, -24.5]	-1.00
Eng	stale reads, crash: R1 vs M1	8/9	0.198	0.790	8	1.000	+0 [-0.00352, +0]	-0.22
Eng	stale reads, restart: M2 vs R2	7/6	0.731	1.000	5	0.250	-0.14 [-0.382, -0.0351]	-0.14
H3	stale reads, warm: P2 vs P1	32/32	<0.001	<0.001	27	<0.001	-4.48 [-4.53, -4.35]	-1.00
H3	p99 latency, warm: P2 vs P1	32/32	0.952	1.000	27	1.000	+0.582 [-0.549, +1.37]	-0.01
H3	DB load, warm: P2 vs P1	32/32	<0.001	<0.001	27	<0.001	+122 [+117, +133]	+0.70
H4	stale reads, M1: partition vs crash	5/9	0.002	0.012	5	0.438	+4.65 [+3.49, +4.75]	+1.00
H4	stale reads, R1: partition vs crash	3/8	0.003	0.017	3	–	+4.67 [+3.49, +4.69]	+1.00
H5	stale reads, app crash: P3 vs P2	5/4	0.016	0.063	4	–	-1.69 [-2.26, -1.4]	-1.00
H5	stale reads, crash: P3 vs P2	2/8	0.533	0.533	2	–	–	+0.38
H5	undelivered, app crash: P3 vs P2	5/4	0.011	0.054	4	–	-67.5 [-80, -66]	-1.00
H5	write p99, no fault: P3 vs P1	2/5	0.095	0.286	2	–	–	+1.00
H5	fill races, no fault: P3 vs P1	2/5	0.206	0.412	2	–	–	+0.50

[6] S. Barahmand and S. Ghandeharizadeh. BG: a benchmark to evaluate interactive social networking actions. In *CIDR*, 2013.

[7] N. Bronson et al. Metastable failures in distributed systems. In *HotOS*, 2021. doi:10.1145/3458336.3465286.

[8] S. Chen et al. Towards scalable and reliable in-memory storage system: a case study with Redis. In *IEEE Trustcom*, 2016. doi:10.1109/trustcom.2016.0255.

[9] S. Ghandeharizadeh and J. Yap. Gumball: a race condition prevention technique for cache augmented SQL database management systems. In *DBSocial*, 2012. doi:10.1145/2304536.2304537.

[10] S. Ghandeharizadeh et al. Strong consistency in cache augmented SQL systems. In *Middleware*, 2014. doi:10.1145/2663165.2663318.

[11] S. Ghandeharizadeh and H. Nguyen. A comparison of two cache augmented SQL architectures. In *TPCTC 2018*, LNCS, Springer, 2019. doi:10.1007/978-3-030-11404-6_8.

[12] L. Huang et al. Metastable failures in the wild. In *OSDI*, 2022.

[13] Z. Jiang et al. Hybrid-Memcached: a novel approach for Memcached persistence optimization with hybrid memory. *IEEE Trans. Computers*, 73(7), 2024. doi:10.1109/tc.2024.3385279.

[14] K. Kingsbury. Elle: finding isolation violations in real-world databases. In *PODC*, 2021. doi:10.1145/3465084.3467483.

[15] Z. Mao et al. Rethinking the cost of distributed caches for datacenter services. In *HotNets*, 2025. doi:10.1145/3772356.3772388.

[16] V. J. Marathe et al. Persistent Memcached: bringing legacy code to byte-addressable persistent memory. In *HotStorage*, 2017.

[17] R. Nishtala et al. Scaling Memcache at Facebook. In *NSDI*, 2013.

[18] C. Richardson. *Microservices Patterns: With Examples in Java*. Manning, 2018.

[19] A. Papaioannou and K. Magoutis. Addressing the read-performance impact of reconfigurations in replicated key-value stores. *IEEE Trans. Parallel Distrib. Syst.*, 33(9), 2022. doi:10.1109/tpds.2021.3135137.

[20] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, 2010.

[21] V. R. Soni et al. Consistency knobs in practice: latency, availability, and staleness across Cassandra and MongoDB under workload and faults. In *IRASET*, 2026. doi:10.1109/iraset68627.2026.11538771.

[22] F. Sun. Evaluating performance and fault tolerance of a linearizable distributed cache in comparison to Redis Cluster. *Elaris Computing Nexus*, 2025. doi:10.65148/ecn/2025004.

[23] A. Vattani et al. Optimal probabilistic cache stampede prevention. *PVLDB*, 8(8), 2015. doi:10.14778/2757807.2757813.

[24] S. Ghandeharizadeh and H. Nguyen. Design, implementation, and evaluation of write-back policy with cache augmented data stores. *PVLDB*, 12(8), 2019. doi:10.14778/3324301.3324302.

[25] H. Zhang et al. Efficient in-memory data management: an analysis. *PVLDB*, 7(10), 2014. doi:10.14778/2732951.2732956.

A Run-level results

Tables 10 and 11 list the per-run values behind every median in the paper; run r of every cell used the same seed. Run counts vary by cell; Table 3 reports the number of valid runs in each experiment, and these two tables the individual observations.

Table 10: Run-level persistent stale reads after recovery (% of reads), runs 0–9.

Fault	Config	Prot.	Outage	Writes	Keys	0	1	2	3	4	5	6	7	8	9
None	M1	P1	5 s	5%	10k	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00	
None	R1	P1	5 s	5%	10k	0.00	0.00	0.00	0.00	0.00					
None	R1	P3	5 s	5%	10k			0.00		0.03					
Crash	M1	P1	5 s	5%	10k	0.00	0.00	0.00	0.00	0.01	0.00		0.00	0.00	0.00
Crash	M1	P2	5 s	5%	10k	0.00	0.00	0.09	0.00	0.00	0.00		0.07		0.04
Crash	R1	P1	5 s	5%	10k	0.00	0.00	0.00	0.00	0.00	0.00		0.00		0.00
Crash	R1	P2	5 s	5%	10k	0.00	0.00	0.00	0.14	0.00	0.03			0.00	0.04
Crash	R2	P1	5 s	5%	10k	4.63	4.67	3.87	4.66	4.64	5.61	3.85	4.32	3.76	4.84
Crash	R2	P2	5 s	5%	10k	0.10	0.17	0.10	0.06	0.14	0.17		0.11		0.08
Crash	R2	P3	5 s	5%	10k	0.12	0.15								
Crash	R3	P1	5 s	5%	10k	4.59	4.64	3.49	4.79	4.67	5.55	3.88		3.99	4.86
Crash	R3	P2	5 s	5%	10k	0.23	0.05	0.05	0.32	0.10		0.07	0.13	0.09	0.04
Restart	M2	P1	5 s	5%	10k	4.37	4.33	3.84	4.55	4.52	5.50	3.65			
Restart	M2	P2	5 s	5%	10k	0.03	0.04	0.05	0.05	0.05	0.07		0.04	0.07	
Restart	M2	P3	5 s	5%	10k		0.04			0.10					
Restart	R2	P1	5 s	5%	10k	4.51	4.52	3.87	4.93	4.58				3.72	
Restart	R2	P2	5 s	5%	10k	0.09	0.09	0.14	0.11	0.10	0.14				0.08
App state loss	R2	P2	5 s	5%	10k	1.75	2.32	1.53		1.78					
App state loss	R2	P3	5 s	5%	10k	0.05	0.06	0.14	0.05	0.10					
Partition	M1	P1	5 s	5%	10k	4.57	4.65	3.49	4.75	4.70					
Partition	M1	P2	5 s	5%	10k		0.08	0.06		0.05					
Partition	R1	P1	5 s	5%	10k		4.69	3.49	4.67						
Crash	M1	P1	5 s	5%	100k	0.00		0.00							
Crash	R2	P1	5 s	5%	100k		3.36			4.38					
Crash	M1	P1	5 s	1%	10k	0.00	0.00	0.00	0.00	0.00					
Crash	M1	P1	5 s	20%	10k			0.04	0.06	0.05					
Crash	R2	P1	5 s	1%	10k	5.29									
Crash	R2	P1	5 s	20%	10k	3.88	3.83		4.42	4.25					
Crash	R2	P2	5 s	1%	10k	0.05	0.05	0.06	0.03						
Crash	R2	P2	5 s	20%	10k	0.14	0.16	0.13	0.10						
Crash	R2	P1	2 s	5%	10k	2.22	2.41	2.27	2.37	1.85					
Crash	R2	P1	10 s	5%	10k	7.74	8.10	7.22	8.34	8.36					

Table 11: Run-level net extra DB queries after recovery, runs 0–9.

Fault	Config	Prot.	Outage	Writes	Keys	0	1	2	3	4	5	6	7	8	9
None	M1	P1	5 s	5%	10k	135	108	73	115	153	343	-40	267	-8	
None	R1	P1	5 s	5%	10k	134	108	41	129	157					
None	R1	P3	5 s	5%	10k			41		146					
Crash	M1	P1	5 s	5%	10k	5,247	5,167	5,253	5,273	5,296	5,478		5,371	5,181	5,280
Crash	M1	P2	5 s	5%	10k	5,247	5,157	5,257	5,277	5,288	5,467		5,406		5,310
Crash	R1	P1	5 s	5%	10k	5,266	5,152	5,269	5,289	5,288	5,477		5,382		5,356
Crash	R1	P2	5 s	5%	10k	5,260	5,184	5,278	5,305	5,312	5,508			5,069	5,303
Crash	R2	P1	5 s	5%	10k	189	161	107	164	212	378	29	303	53	277
Crash	R2	P2	5 s	5%	10k	282	293	226	311	330	541		425		415
Crash	R2	P3	5 s	5%	10k	314	328								
Crash	R3	P1	5 s	5%	10k	202	122	63	114	162	344	-6		-10	243
Crash	R3	P2	5 s	5%	10k	293	238	185	235	279		117	412	129	364
Restart	M2	P1	5 s	5%	10k	195	186	96	185	225	382	32			
Restart	M2	P2	5 s	5%	10k	328	281	223	371	330	522		414	167	
Restart	M2	P3	5 s	5%	10k		313			317					
Restart	R2	P1	5 s	5%	10k	174	165	104	153	211				74	
Restart	R2	P2	5 s	5%	10k	325	281	194	309	310	514				406
App state loss	R2	P2	5 s	5%	10k	250	250	211		281					
App state loss	R2	P3	5 s	5%	10k	330	329	254	307	329					
Partition	M1	P1	5 s	5%	10k	136	109	57	119	158					
Partition	M1	P2	5 s	5%	10k		231	171		277					
Partition	R1	P1	5 s	5%	10k		112	46	118						
Crash	M1	P1	5 s	5%	100k	10,094		10,201							
Crash	R2	P1	5 s	5%	100k		293			279					
Crash	M1	P1	5 s	1%	10k	5,414	5,529	5,535	5,581	5,517					
Crash	M1	P1	5 s	20%	10k			4,244	4,253	4,743					
Crash	R2	P1	5 s	1%	10k	104									
Crash	R2	P1	5 s	20%	10k	480	703		192	709					
Crash	R2	P2	5 s	1%	10k	123	221	91	179						
Crash	R2	P2	5 s	20%	10k	869	935	574	516						
Crash	R2	P1	2 s	5%	10k	216	162	71	213	221					
Crash	R2	P1	10 s	5%	10k	184	133	39	91	230					