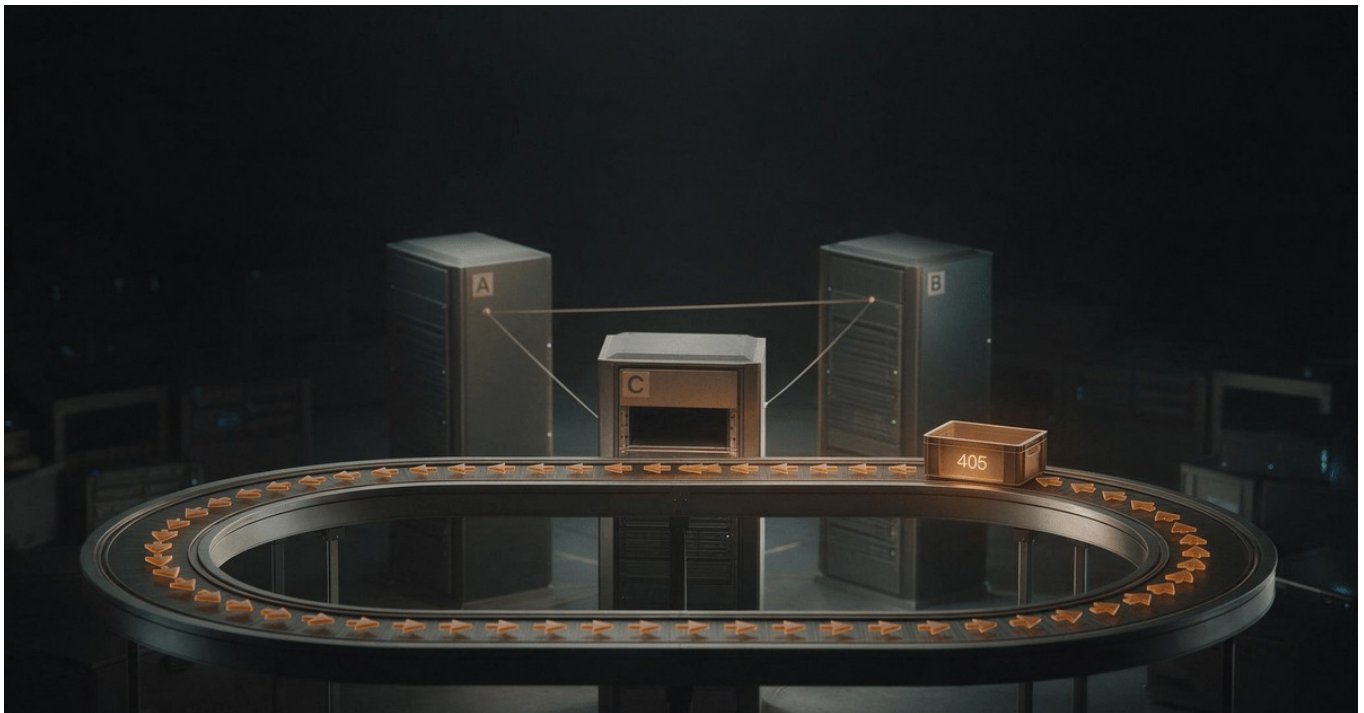


MinIO delete-marker convergence: how the Silo fork fixed the 405 replication loop

Julien Laurenceau | Published 2026-09-15 | Updated 2026-09-30

<https://pepitedata.com/minio-silo-delete-marker-convergence-fix/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



Working with me: this is the kind of review I run for clients. The [MinIO, S3 and object storage consulting](#) page describes what an independent cluster audit covers, from architecture and multi-site replication to performance and sizing.

In a recent post, I documented how a MinIO replication loop turned ILM expiration into 706 million replication requests in 48 hours, 96.8 percent of them answered 405, carrying no data. Earlier, I documented a drop counter MinIO increments in four places and reads in none. Both analyses ended the same way: the convergence path is a code fix, and no such fix existed anywhere.

That changed. I filed the two findings on the tracker of [Silo](#), an actively maintained MinIO fork, as [issue #152](#) (the MRF queue drops nobody can see) and [issue #153](#) (the delete-marker loop behind the 405 storm). The fork merged a first fix, [PR #162](#), within days. Reviewing that fix against my own three-site reproduction surfaced three defects it did not cover, the *retry* side of the same loop, and that second fix is now up as [PR #184](#), with a design document checked into the repo so the mechanism outlives the incident.

This post tells that story: what the three root defects actually are, what the first fix got right (and why it deliberately rejected my own first suggestion), what the second fix adds, and how the whole thing is verified. It is the closing chapter of the 405 storm investigation.

The loop, reduced to three defects

The loop reduces to three defects: a retry shortcut that suppresses purge retries, a success recorded in the wrong status field, and an MRF heal pass that drops every delete marker. This is the recap of the mechanism from the [405 storm post](#), since everything below hangs on it. A delete marker in site replication carries two independent per-target status maps:

- Targets, the marker's **creation** status per peer (arn1=COMPLETED;arn2=COMPLETED;)
- PurgeTargets, the **purge** status per peer (arn1=PENDING;arn2=PENDING;), written when an ILM flow or a versioned DELETE `?versionId=` purges the marker

On the wire, creation and purge travel as the *same* request shape (DELETE `?versionId=X` with `x-minio-source-deletemarker: true`). And the replication engine probes the target with a versioned StatObject that *expects* a 405, Method Not Allowed, as the normalized "this version exists and is a delete marker" answer.

Three defects then combine into a permanent loop:

#	Where	Defect	Consequence
1	replicateDeleteToTarget, the early-out	The "already replicated" shortcut tested the marker's creation status, even for a purge	A purge whose first delivery failed was never retried , so the peer keeps the marker forever
2	replicateDeleteToTarget, outcome branches	A successful purge recorded <code>ReplicationStatus = Completed</code> instead of <code>VersionPurgeStatus = Complete</code>	The source's purge state stays PENDING forever, the scanner re-queues it every cycle, and the purged version never leaves <code>xl.meta</code>
3	queueMRFHeal	GetObjectInfo on a marker version deterministically answers 405, and the MRF dropped the entry on any error	The fastest retry path never ran for deletes; recovery fell back to the slow scanner

With two sites, the first fan-out attempt almost always succeeds and the data converges despite the broken bookkeeping. With three or more peers per bucket, at least one first-attempt delivery eventually fails, and then defects 1+2 make recovery impossible while defect 3 removes the fast path. In my 84-million-version mesh: about 20 million versions stuck in PENDING from the day the ILM rules shipped, and a model that predicts about 685 million useless requests per 48 hours, matching the measurement.

What landed first: PR #162, the first attempt

The fork's first fix (PR #162, merged September 9) attacked the problem I had under-weighted: the *first* delivery of a marker purge. In stock code, the single-object DELETE `?versionId=` handler classified any delete marker, including one carrying purge state, as a marker *creation*. The purge then rode the marker path, where the source's purge status never got finalized. PR #162 reclassified it:

```
if objInfo.DeleteMarker && objInfo.VersionPurgeStatus.Empty() {
    dmVersionID = objInfo.VersionID // marker creation: fan out as before
} else {
    versionID = objInfo.VersionID // version purge: first attempt finalizes it
}
```

With that, the version-purge path (which already recorded VersionPurgeStatus correctly) handles the common ILM case end to end, and the heal path learns to recover markers left PENDING by older

versions.

The same PR did two other things my posts had flagged. For #152: the [write-only MRF drop counters](#) (TotalDroppedCount and TotalDroppedBytes, incremented in four places, read in none; droppedCount_since_uptime always reported 0) are now published through the admin snapshots **and** both Prometheus metric trees, with overflow logged under the default replication_priority=auto instead of silently deduped away. The misrouted getWorkerCh arguments (object name and bucket swapped on the delete path) got aligned too. And it hardened site-resync cancellation: each resync now owns a cancelable context, so mc admin replicate resync cancel cannot strand a walk that dispatches into a full worker queue.

One decision deserves its own paragraph. Issue #153 had suggested making the 405 probe *terminal* for purge-carrying markers: "we saw a 405, the marker is there, mark the purge complete and move on." The fork rejected that, and it is the right call. **A present marker is not proof of a completed permanent deletion.** The 405 says the marker version exists *right now*; a purge that returns "complete" on that evidence can leave the marker on the peer while the source claims convergence. Instead, PR #162 fixed the *classification* so the first attempt genuinely purges, and kept 405 meaning exactly what it means. The bug report was correct about the mechanism and wrong about the fix; being overruled with a better argument is what a good review is for.

What the first fix did not cover: the retry side

The retry side is what the first fix did not cover, because replication is never one attempt. A purge is retried by the scanner heal every cycle while its composite status is not Complete, and failed replications re-enter through the MRF. Those retry paths were still intact, and still broken in three ways.

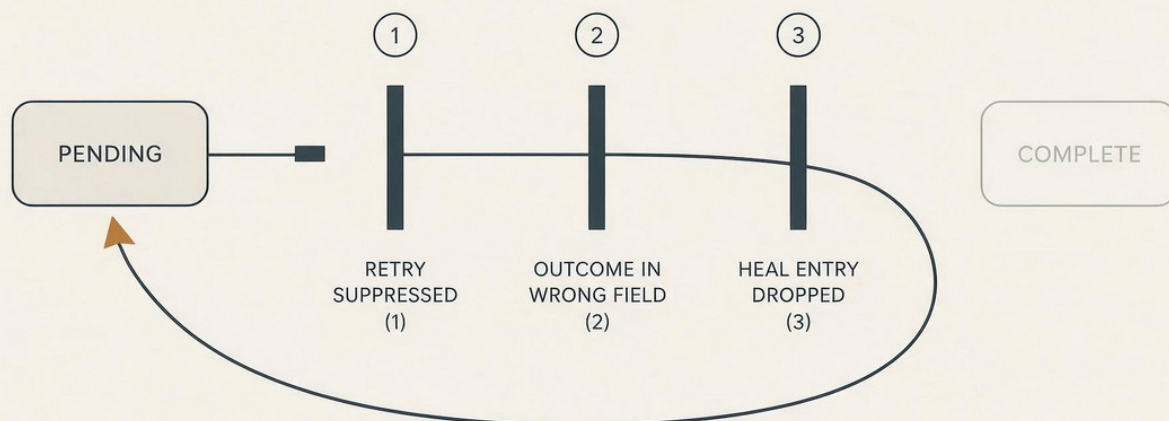
Bug 1: the early-out ate purge retries. The retry shortcut was written for marker *creation* ("already replicated, do not re-send"):

```
if dobj.VersionID == "" && rinfo.PrevReplicationStatus == replication.Completed && ... {
    rinfo.ReplicationStatus = rinfo.PrevReplicationStatus
    return rinfo
    // a purge never gets sent
}
```

Both creation and the legacy purge shape travel with dobj.VersionID == "" (the purged version rides in DeleteMarkerVersionID). For a purge retry, PrevReplicationStatus is the marker's old **creation** status, COMPLETED on every site that ever received the marker. The purge was returned as Completed without ever being sent. Concretely, on a three-site mesh:

```
A purges marker X:  PurgeTargets={B:PENDING, C:PENDING}
fan-out #1:         B purged OK    C network error FAIL
                   PurgeTargets={B:COMPLETE, C:FAILED}
scanner heal:       B: early-out fires (creation COMPLETED) -> no-op
                   C: early-out fires (creation COMPLETED) -> NO-OP, C keeps marker X
                   composite purge stays FAILED, re-queued next cycle, forever
```

Only an explicit mc admin replicate resync bypassed the early-out, which is why operators in storm situations keep re-running resyncs, feeding the 405 traffic.



The cycle the three retry-side defects produce: the purge state never advances past the gates, so COMPLETE stays unreachable and the re-queue loop runs forever.

Bug 2: success recorded in the wrong field. When the purge *did* get sent and succeeded:

```

} else {
  if dobj.VersionID == "" {
    rinfo.ReplicationStatus = replication.Completed // wrong field for a purge
  } else {
    rinfo.VersionPurgeStatus = replication.VersionPurgeComplete
  }
}

```

`rinfo.VersionPurgeStatus`, the field `getReplicationState` persists into `VersionPurgeStatusInternal`, kept its seeded `PENDING`. So even a fully successful purge left the source metadata claiming otherwise: the scanner re-queued it every cycle, and the final `xl.meta` rewrite never took the purge-complete removal path, so the marker version lingered on disk indefinitely. PR #162's own regression test documented this behavior as expected: the legacy-shaped purge was asserted to stay `PENDING` and wait for a heal. That is convergence eventually, not convergence.

Bug 3: the MRF dropped every delete marker. Failed replications are saved to the MRF with `versionID = DeleteMarkerVersionID` for markers. The heal pass then calls `GetObjectInfo(VersionID=X)` on the source, which answers **405 MethodNotAllowed with valid object info**, and the entry died on a bare `if err != nil { continue }`. Every failed delete replication silently lost its fastest retry path and waited for the slow scanner walk.

The fix: classify the purge from its own state

PR #184 (in review) closes the three with one precise classification at the top of `replicateDeleteToTarget`: a purge carries purge state and a creation does not. That classification settles three consequences mechanically: retries are no longer suppressed, outcomes land in the purge field, and the MRF heals markers.

```

isDMPurge := dobj.VersionID == "" &&
  dobj.DeleteMarkerVersionID != "" &&
  !rinfo.VersionPurgeStatus.Empty() // a purge carries purge state; a creation doesn't

```

Three consequences, all mechanical:

- 1. **Retries are no longer suppressed.** The early-out gains a `!isDMPurge` guard, and for a purge the creation status is pinned (`rinfo.ReplicationStatus = rinfo.PrevReplicationStatus`) so retries neither get eaten by it nor overwrite it.
- 2. **Outcomes land in the purge field.** The offline, error and success branches write `VersionPurgeStatus` whenever `dobj.VersionID != "" || isDMPurge`. The composite reaches `Complete`, the source rewrite takes the version-removal path in `DeleteVersion`, and the purged marker version is actually dropped from `xl.meta`.
- 3. **The MRF heals markers.** The heal pass accepts the 405, which carries a fully populated `ObjectInfo` for the marker version, and routes it back through the delete replication path:

```
if err != nil && !isErrMethodNotAllowed(err) {
    continue
}
if oi.Name == "" {
    continue
}
QueueReplicationHeal(p.ctx, e.Bucket, oi, e.RetryCount)
```

The behavior table, before and after:

Scenario	Before	After
Purge, first delivery fails to site C	C keeps the marker forever	retried by scanner and MRF until C confirms
Purge, all deliveries succeed	source purge state stays PENDING forever; version lingers; re-queued every scan	status COMPLETE, version dropped, queue drains
Legacy marker-path purge (pre-#162 state)	left PENDING, needed a heal re-queue	converges on the first proper delivery
Marker creation retry, target ready	405 probe then COMPLETED	unchanged, by design

Note what does *not* change: transient 405s during a single heal round are still there. A 405 that means "already replicated" once per retry is the protocol working. The storm was the *unbounded repetition* driven by states that could never advance; that is what disappears. The general rule from #153 survives intact: never re-enqueue on a deterministic response; only transient errors (network, 503 SlowDown, quorum loss) justify re-queueing.

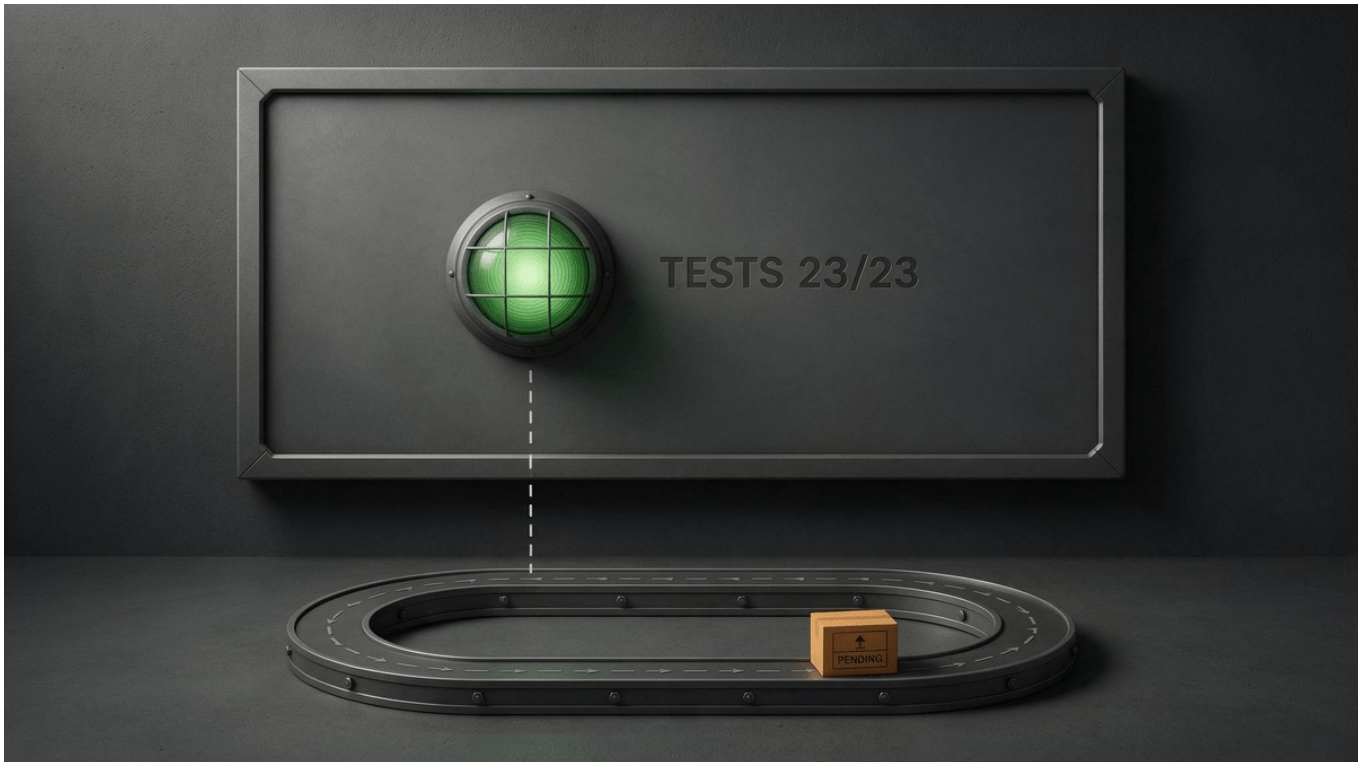
The fix also matters beyond site replication: `replicateDeleteToTarget` is the shared delete engine under bucket replication too, so any deployment whose replication topology is large enough to see a failed first delivery gets the same convergence back.

Testing convergence, not just code paths

The tests in this PR prove that the state converges, not just that the code paths run: the marker version gone from source and target, purge status `COMPLETE`, and nothing further scheduled for healing. The interesting part of a fix like this is not the diff; it is proving the *state* converges. Two test layers in the PR:

- **Target semantics, table-driven.** `replicateDeleteToTarget` against a scripted HTTP peer: a purge in the legacy shape, seeded with creation status `COMPLETED` and purge status `PENDING`, must not short-circuit (the remote must observe the `DELETE`), must end with `VersionPurgeComplete` on success and `VersionPurgeFailed` on rejection, and must leave the creation status untouched in both cases.

- **End-to-end legacy recovery.** The erasure-layer test that #162 used to document the old behavior is reworked: a marker seeded in the legacy producer's shape is purged through the full replicateDelete path, and the assertions are now *convergence*: the marker version gone from source and target, purge status COMPLETE, and nothing further scheduled for healing.



Green tests are not convergence. The reworked assertion checks that the marker version is actually gone from source and target, not that the code path was exercised.

The whole replication, MRF and resync surface (23 tests) plus the delete-handler suites pass unchanged, which is the point: the fix narrows the defect without moving the semantics #162 deliberately preserved.

Why a fork, and what upstream can take back

Silo is the MinIO fork maintained by the Pigsty project, and it has become the pragmatic place where community-found replication bugs get fixed without waiting on upstream triage, and it now sits at the center of the release-channel decision every Community operator faces (I assess that separately in [MinIO Community in Production](#)). The pattern that made this work is worth naming, because it is the part other forks usually get wrong:

- **Every fix lands with a reproduction-anchored regression test.** The test that used to *document* the broken behavior is the one that now *fails without* the fix.
- **The mechanism is written down in the repo.** [docs/site-replication/delete-marker-convergence.md](#) carries the full analysis: wire format, state bookkeeping, the three defects, the storm anatomy, a three-site mc reproduction script, and the known remaining limitations. Six months from now, nobody has to re-derive any of it from `git blame`.
- **The patch stream stays reviewable.** #162 is one PR with three separable parts; #184 touches one function plus one heal pass. Upstream MinIO can cherry-pick either without dragging fork-specific infrastructure along.

What remains open, honestly, is in the limitations section of that doc: the wire protocol still has no tombstone for a removed marker (a resync from a stale site can *re-create* a purged marker), a fan-out

that skips an offline target silently composites to COMPLETED, replica markers are dead-ended, and the per-response-code observability I asked for in #153 (a 405 counter, a gauge of stuck versions) is still a follow-up. The storm's *cause* is closed; the instrumentation that would have caught it early is not built yet.

Status: #152 and #153 are closed by PR #162, merged 2026-09-09. The convergence follow-up, PR #184, is open and in review.

FAQ

Why not just make the 405 terminal, as the original bug report suggested?

Because a 405 proves the marker is *present*, not that the purge *succeeded*. Treating presence as completion can mark a purge complete while the marker still sits on the peer, which is silent divergence with a green status. The correct shape is what landed: make the first attempt a real version purge, record outcomes in the purge status field so retries converge, and let 405 keep its meaning. The loop dies because the states advance, not because the probe was silenced.

I run stock MinIO CE and I am seeing the storm. What can I do today?

Drain the stock: purge the affected objects on the source (`mc rm --versions --force`, batched) so their share of the scanner walk drops to zero. The [previous post](#) details the procedure and its caveats. Slow the scanner to protect the client SLA while you do it; it divides the storm, it does not fix it. Phase your ILM expiration rules and keep the replication topology small: with N sites per bucket, every failed first delivery builds a permanent PENDING. And track the fixes: the vendor's public changelog is the [AIStor release notes index](#); upstream has nothing equivalent merged at the time of writing.

Does the #184 fix change replication semantics?

No, it removes three ways the *existing* semantics could never converge. Marker creation retries still treat 405 as "already replicated". Purges still require an explicit peer confirmation (a successful `RemoveObject`, or a 404 on a later probe meaning the marker is gone). What changed is bookkeeping: purge outcomes now land in the purge field, so the scanner and the MRF stop re-queueing work whose state can never advance.

How do I verify convergence on my own cluster after upgrading?

Three signals: `mc admin replicate status` per site shows purge statuses reaching COMPLETE instead of camping on PENDING or FAILED; the 405 share of replication traffic (measure it with `mc admin trace` filtered on the internal replication user-agent) decays to occasional single-round probes instead of a floor that never moves; and the MRF drop counters exposed since #162 no longer climb during quiet periods. A stuck-version gauge is still a follow-up. For now, a scheduled `mc ls --versions diff` across sites on a sample of prefixes is the honest check.

Is bucket replication affected too?

Yes, for the better. `replicateDeleteToTarget` is the shared delete engine under both bucket and site replication, so bucket-replication deployments with enough objects to see a failed first delivery (which is everyone, eventually) get the same retry and status-recording corrections.

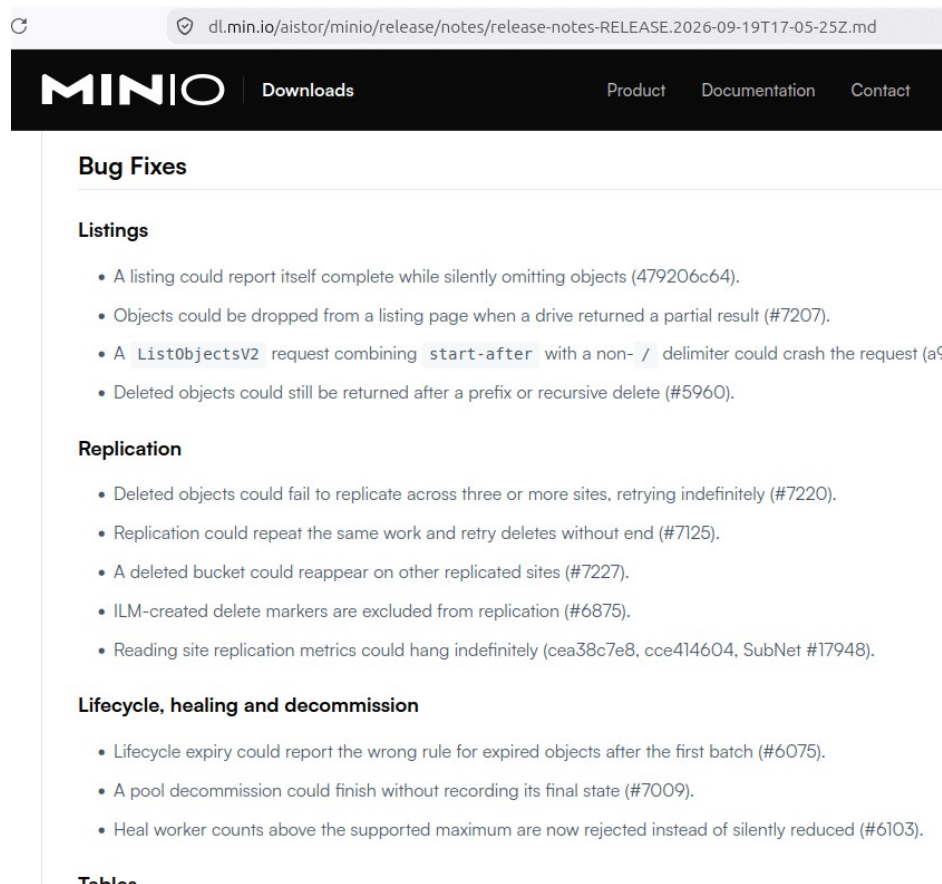
Does the AIStor release line fix these replication bugs?

The [AIStor release notes for RELEASE.2026-09-19](#) list three fixes in the same defect class as the three defects documented above: delete replication retrying without end, ILM delete markers excluded from

replication, and deletes failing to replicate across three or more sites. The notes describe the symptoms in the vendor's own words; whether the implementation matches Silo's PR #184 is not something a release note settles. The entries, verbatim:

- Deleted objects could fail to replicate across three or more sites, retrying indefinitely (#7220).
- Replication could repeat the same work and retry deletes without end (#7125).
- ILM-created delete markers are excluded from replication (#6875).

The same release also notes that delete replication no longer floods logs with repeated messages, and that reading site replication metrics no longer hangs indefinitely. These notes live on the vendor's download host and can disappear; the screenshot below is the durable copy.



The vendor's own bug-fix list for RELEASE.2026-09-19: three of the five replication entries match the defects documented in this post.

Update 2026-09-29. This post is now part of [Data Platform Resilience: RTO, RPO and Replication in Practice](#), the field guide to RTO, RPO and replication.

Related posts

- [Data Platform Resilience: RTO, RPO and Replication in Practice](#)
- [MinIO site replication bug: how 684 million HTTP 405s exposed the delete-marker loop - the mechanism and the measurements this post closes out.](#)
- [MinIO replication drops objects into a counter nobody reads - the MRF drop observability that #162 finally wired up.](#)
- [MinIO Community in production: assess the risk you are running - the release-channel decision this fork now sits in.](#)

- [MinIO Site Replication: Sync Mode Will Not Save Your RPO](#) - the replication architecture side of the same platform.
- [MinIO and Lots of Small Files: Why It's Not a Database](#) - why object count and the scanner dominate operations.

If you run a MinIO, Ceph or S3 platform where replication behavior nobody can explain is eating your CPUs, that is exactly the kind of cross-layer investigation I run in a [data platform performance audit](#). [Book a 15-min intro call](#) to see which engagement fits, or take one focused hour with an [Expert Call](#).

PDF snapshot of the public article. The online version is authoritative; links remain clickable.