

MinIO site replication bug: how 684 million HTTP 405s exposed the delete-marker loop

Julien Laurenceau | Published 2026-09-06 | Updated 2026-09-30

<https://pepitedata.com/minio-site-replication-405-delete-markers/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



Update, September 2026: the story closed. The fork merged a first fix on 2026-09-09, and my follow-up pull request completing the retry-side convergence is now written up in full: the three defects, the rejected-and-right design call, and the convergence tests. Read it here: [MinIO delete-marker convergence: how the Silo fork fixed the 405 replication loop](#).

Update, September 2026: the open-source line has a maintained fork now, PGSTY Silo, and this bug was reported against it. The maintainer merged a first fix on 2026-09-09, and my follow-up pull request completing the purge-retry and MRF-healing convergence is under review. What the fork changes for Community operators, and where it fits in the release-channel decision: [MinIO Community in Production: Assess the Risk You Are Running](#). If you run MinIO Community or the Silo fork in production, an independent review of that layer is a service I sell: [MinIO, S3 and object storage consulting](#).

Two days into a replication storm on a production MinIO platform, the measurement finally landed: 706 million replication requests in 48 hours, 96.8 percent of them answered with 405, carrying no data at all. The cluster was burning its CPUs attempting to replicate, failing deterministically, and retrying forever. This post describes the mechanism as I reconstructed it in the open source code, the numbers that pin it down, the fix directions, and the feature-surface advice I give in the meantime.

The facts

A platform running a site replication mesh (three sites), LOSF profile: millions of small objects, 84M versions on the counter. One day, ILM expiration rules are deployed. The same day, the drift begins: CPU at 100%, memory climbing, and a replication query storm that never stops.

Two days into the storm, with the platform finally instrumented end to end, the measurement lands:

Measurement (48 h)	Value
Total replication requests	706.3M (about 4,100 qps)
of which 405 responses	684M (96.8%)
of which 2XX responses	22.3M (3.16%)
Concurrent client traffic	about 2,000 qps (95% HeadObject)
Cluster CPU	about 60%, with 0.5% iowait

In other words: replication traffic weighs twice the client traffic, it is composed of 96.8% 405 responses, and it carries **no data**. The cluster burns its CPUs attempting to replicate, failing, and retrying.

The first suspect is easy to name: the scanner walks the namespace faster than the replication queue drains, so objects get re-queued perpetually. Digging into the code, that hypothesis is right, but the precise mechanism is more insidious: it involves delete markers, a protocol signal that uses the 405 status code, and an overly restrictive exit condition.

The 405 is not an error: it is the delete-marker signal

In S3, a HEAD on a version that is a delete marker returns neither 200 nor 404: it returns **405 Method Not Allowed**. That is the normalized way of saying "this version exists, and it is a delete marker" on a HEAD request.

MinIO replication deliberately builds on this semantics. To replicate a delete marker to a peer, the source issues a versioned StatObject against the target with an internal header, and **expects the 405**:

```
// cmd/bucket-replication.go, replicateDeleteToTarget (OSS master, L645-664)
toi, err := tgt.StatObject(ctx, tgt.Bucket, dobj.ObjectName, minio.StatObjectOptions{
    VersionID: versionID,
    Internal: minio.AdvancedGetOptions{
        ReplicationProxyRequest: "false",
        IsReplicationReadyForDeleteMarker: true,
    },
})
serr := ErrorRespToObjectError(err, dobj.Bucket, dobj.ObjectName, dobj.VersionID)
switch {
case isErrMethodNotAllowed(serr):
    // delete marker already replicated
    if dobj.VersionID == "" && rinfo.VersionPurgeStatus.Empty() {
        rinfo.ReplicationStatus = replication.Completed
        return rinfo
    }
    ...
}
```

A 405 here means "the delete marker is already present on the target", hence **replication already done**. In resync accounting, a 405 on a delete marker even counts as a success (ReplicatedCount++). This is clean design: the protocol reuses a standard S3 code as an acknowledgment.

Except.

Anatomy of the loop

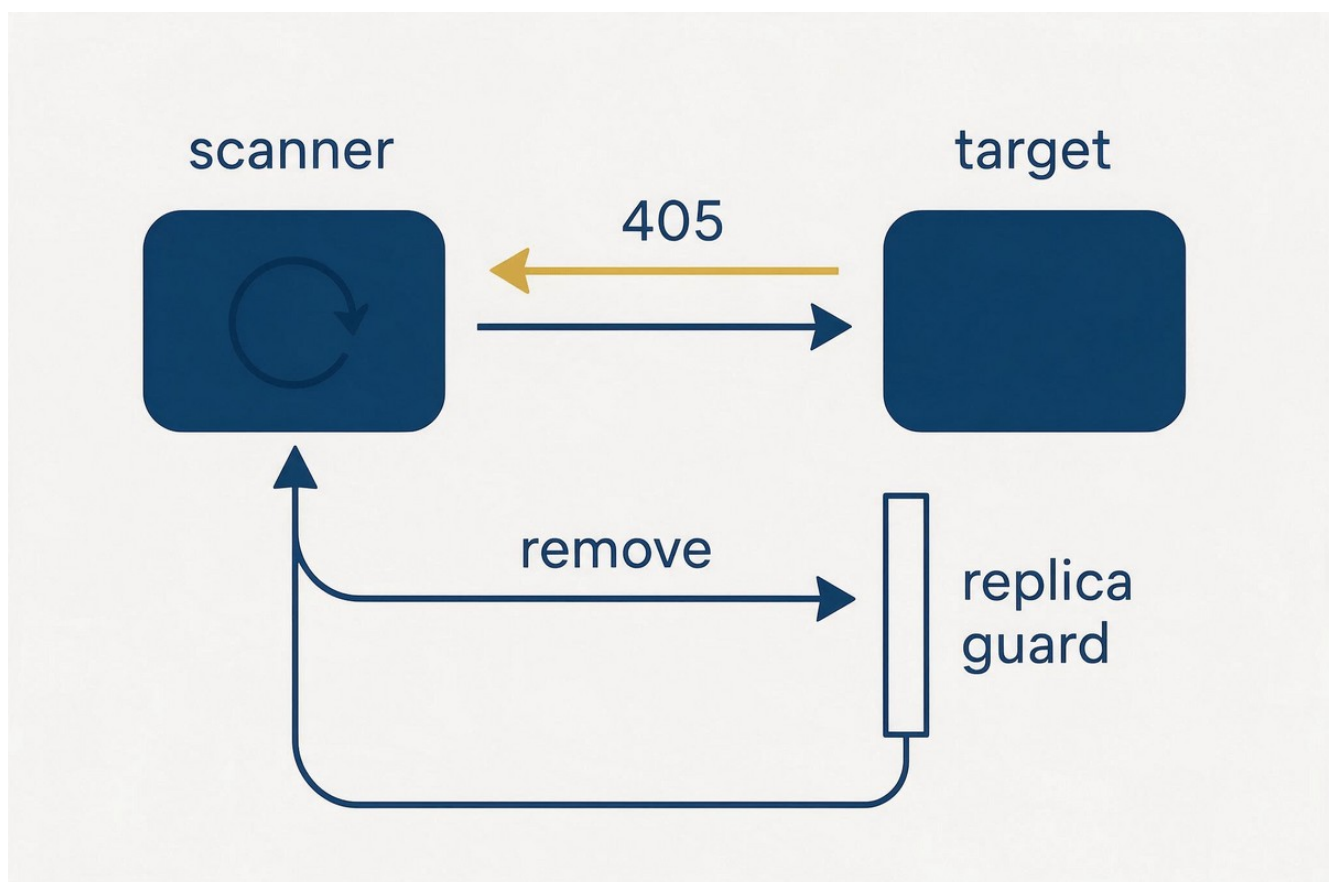
The success exit above is conditioned on two things: `doobj.VersionID == ""` (the operation is a plain delete-marker creation) **and** an empty `VersionPurgeStatus` (no version purge in flight). If either condition fails, the code does not exit: it falls through to the fallback `RemoveObject`, which attempts the deletion on the target.

There exists a population of delete markers that ticks exactly the wrong boxes: **delete markers carrying a version-purge status**, created by ILM flows (`NoncurrentVersionExpiration`, `purge-all-object-versions`, `retention purges`). For those operations:

1. the scanner visits the version (every ~86 minutes in our case, see below);
2. the source probes the target: the target holds the marker, the HEAD answers 405;
3. the `Completed` exit is gated (`VersionID` non-empty or purge status present);
4. the code falls through to a `RemoveObject` toward the target;
5. the target applies its delete-on-replica guard: a version whose replication status is `REPLICA` is not marked deleted by the target (`cmd/erasure-object.go: markDelete = false`; deletions only travel from source to replica, never the reverse). The operation cannot converge;
6. the source-side status stays non-`Completed`, so the object is re-queued;
7. back to step 1.

Loop closed. And the kicker: each 405 of this loop is, from the protocol's point of view, a **success response**. Replication sees no failure, no error counter moves, and the traffic keeps growing as long as the scanner visits faster than the loop closes.

```
scanner visit (each cycle) -> StatObject probe -> 405 ("marker present")
-> Completed exit gated (VersionID / purge status) -> RemoveObject on target
-> replica guard, no-op -> status still non-Completed
-> next scanner visit -> ...
```



The closed loop: a deterministic 405 probe whose Completed exit is gated, a RemoveObject the replica guard blocks, and the version re-queued at the next scanner cycle.

The model that matches the measurements

The model that matches the measurements is a simple product: the stuck stock multiplied by the scanner cycles per day. Three measurements close the account: 84M versions in the namespace, a measured scan rate of 16.3k versions per second that walks the whole namespace in 86 minutes, and about 20M versions stuck in the loop.

- 84M versions in the namespace;
- measured scan rate: 16.3k versions/s, i.e. a full cycle in **86 minutes**;
- about 20M versions stuck in the loop (about 24% of the namespace, a population that appears the day the ILM rules are deployed).

```
replication_queries_per_day ~= stuck_versions * 86,400 / scan_cycle_s
                             ~=
                             20.4M x 16.8 cycles/day
                             ~=
                             343M/day ~= 685M / 48 h
```

The measured 684M 405s land exactly there. The 22.3M 2XX are legitimate replications (fresh writes) that work normally. The success rate of the stuck stock is **zero, measured**: a 405 is deterministic, the same attempt will fail forever. No scanner speed tuning changes that: slowing the scanner divides the storm and the convergence by the same factor.

Why it is insidious, and why it went unnoticed for so long

The bug is insidious and went unnoticed so long because three blind spots stack up: the replication error counters stay silent, the replication queue is blind to its own lag, and the stuck debt never converges. With no functional failure and no error signal, the onset looks like organic growth instead of a bug.

1. **Replication error counters stay silent**: the 405s are protocol success responses. No native alert sees a storm running at 4,100 qps.
2. **The replication queue is blind**: silent drops on overflow and "leaked" queues invisible in metrics. The actual lag cannot be read anywhere.
3. **The debt never converges**: every attempt on the stuck stock fails deterministically. Slowing the scanner reduces the noise but fixes nothing; speeding the scanner amplifies the storm. The only path to convergence is fixing the mechanism itself.

The storm is a scaling phenomenon, not a threshold. The probe rate is simply:

```
probes_per_second = scan_rate x stuck_fraction
```

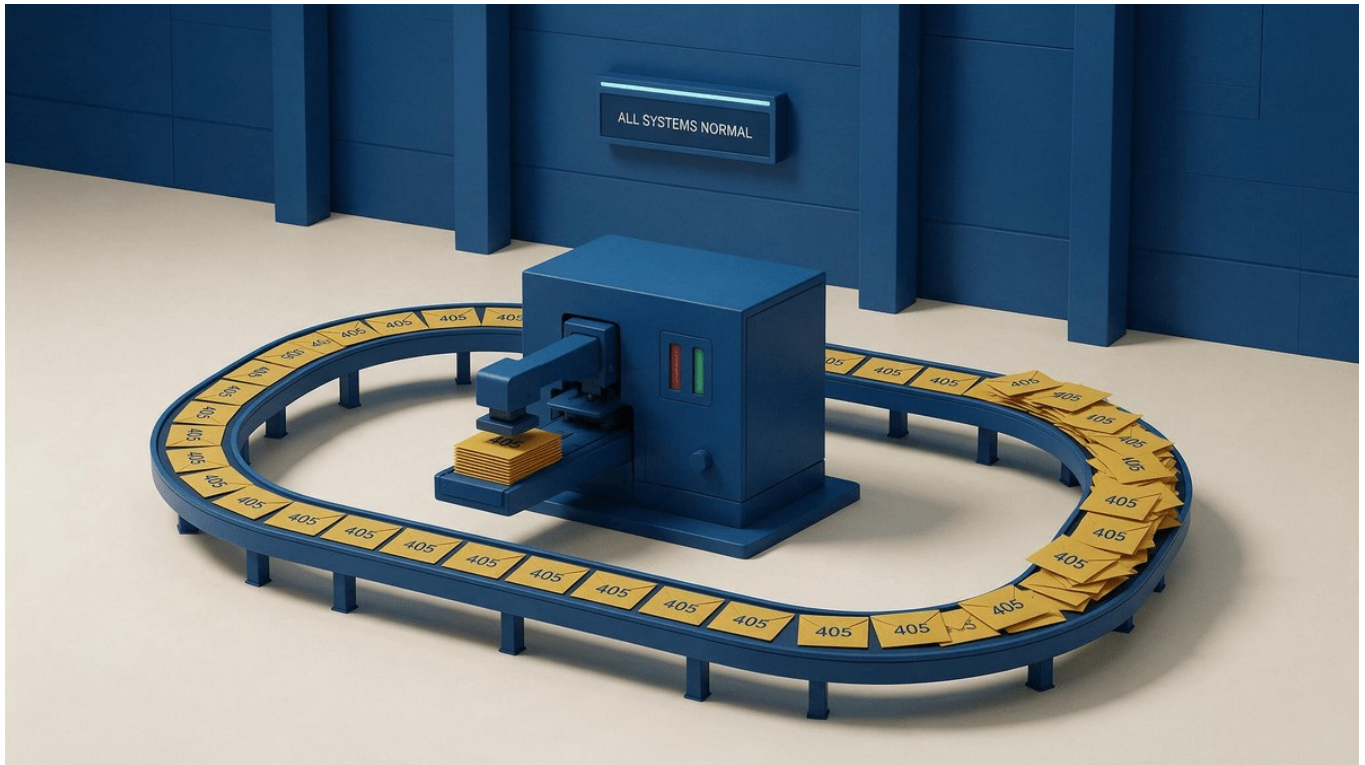
No cliff, no error, no minimum namespace size: the loop exists at any scale, and its cost is the product of two factors that both grow slowly. On a 6M-version namespace with a 480/s scanner and 0.5% of versions stuck in the loop, it costs 2.4 qps: invisible, and arguably running unnoticed for years in many deployments. The same code on a 84M-version namespace with a 16.3k/s scanner and 24% of versions stuck produces about 4,000 qps. Nothing changed in the code between the two; the product of two slow-growing factors did. The scan rate grows with the namespace (operators keep scan cycles reasonable as data grows, and drive counts grow with it), and the stuck fraction creeps up with every ILM purge cycle and every mesh divergence (each site outage, each LWW divergence, adds markers to the loop).

Four ingredients must stack for the storm to matter: site replication, ILM version-purge rules (the purge-status markers), tens of millions of versions, and a fast scan cadence. Remove any one and the symptom is negligible: no site replication, no probes; no ILM purges, markers take the Completed exit and terminate; small namespace or slow scanner, a trivial probe rate. Extreme LOSF platforms with regulatory retention hit the four at once, which is why the bug becomes evident exactly where data protection requirements are the strictest.

And nothing breaks, so nothing calls for attention. The 405 answer is true: the markers are replicated. Listings stay coherent, GET/PUT behave, durability dashboards stay green, no object is lost. The only symptoms are a creeping CPU load and a replication debt whose size nobody can read (capped, blind queue). On a busy cluster, 4,100 qps of cheap 405 rejections is easy to misattribute: client load, large-object replications, scanner work. Without the response-code distribution of the replication traffic, there is no reason to ever look.

Nothing is lost, and nothing deleted is guaranteed gone either. The same blindness has a correctness side that outlives the CPU storm. Every version stuck in this loop is a deletion the mesh never settles: the source considers the version expired, the peers still hold it, and the delete-on-replica guard is exactly what keeps them holding it. On top of the stall, the replication queue drops events silently on overflow, so some deletes are not delayed but lost: the peer never learns the object was deleted, and the object simply stays live there. The loop makes that drop path worse over time: the stuck stock is re-queued at every scanner cycle, the queue stays saturated near its cap, and the entries that overflow into the discard path are increasingly fresh, legitimate ones. I dissected the accounting of that path separately in [why MinIO replication drops objects into a counter nobody reads](#); the two failure modes compound, and the storm this post describes is one of the things that feeds the drops. This is the zombie scenario of site replication. Fail over to a peer, or resync from one, and versions your application deleted come back as accessible data; under last-write-wins, a live peer copy can even win a conflict and travel back to the site that deleted it. No error counter records any of this, and with a capped blind queue nobody can enumerate the delete debt. My earlier note on [why sync mode will not save your RPO](#) made the point architecturally; this loop shows the same gap at the level of a single delete operation. A replication that reports a delete as replicated while its settlement never terminated is a best-effort mirror, and it must be operated, monitored and audited as one.

That is why the bug can run for months: a resource amplification loop with no functional failure, no error signal, and a gradual onset that mimics organic growth. It became evident only when the platform was instrumented end to end and the replication-to-client traffic ratio showed up: two to one, 96.8% of it in 405s. The net result: 60% of CPU consumed by traffic that carries nothing, while the legitimate replication path fights through this noise.



The signature of the storm: every answer says the work is already done, no counter moves, and the board stays green while 96.8% of the replication traffic carries no data and no error.

Fix directions

Direction 1: make the 405 probe terminal for delete-marker ops, whatever the associated purge. The marker's presence on the target *is* the replicated state; the removal of the marker version itself belongs to target-side ILM or a dedicated internal API, not to a re-probe:

```
case isErrMethodNotAllowed(serr):
    // 405 on a versioned HEAD proves the target holds this version as a
    // delete marker: the marker is already replicated. Terminal outcome,
    // otherwise delete markers carrying a purge status (ILM flows) are
    // re-probed forever at scanner cadence.
    rinfo.ReplicationStatus = replication.Completed
    if dobj.VersionID != "" && !rinfo.VersionPurgeStatus.Empty() {
        rinfo.VersionPurgeStatus = replication.VersionPurgeComplete
    }
    return rinfo
```

Direction 2: general principle, never re-enqueue on a deterministic response. A 405 (like the already-terminated 404 probe case) is not "retried with hope": only a state change makes it obsolete. Transient errors (network, 503 SlowDown, quorum) justify re-queueing; deterministic outcomes require a terminal status transition, or the catch-up path becomes an amplifier whose work rate is proportional to the accumulated backlog, with no bound in sight.

Direction 3: the observability that would have saved days. Two counters: total replication requests **per response code** (405 in particular), and the number of versions with a non-Completed replication status older than N hours. Today, neither exists natively, and the capped queue makes the lag unreadable.

In the meantime, on the ops side

Three moves kept the platform standing, in order of effectiveness: measure the response-code distribution of the replication traffic, slow the scanner down to divide the storm at the source, and shape the replication path at the frontend level to protect the client SLA during the investigation. Convergence

stays blocked while they run, which is why they are stopgaps rather than a fix.

- measure first: response-code distribution of the replication traffic (`mc admin trace` filtered on the internal User-Agent, or the logs of the replication frontend); this is what revealed the 96.8% of 405s;
- slow the scanner down (`scanner speed=slowest`): divides the storm at the source, at the price of a slower catch-up, acceptable when convergence is blocked anyway and large replication lag is acceptable;
- shape the replication path at the frontend level (internal traffic to peers goes through the same HAProxy as clients on many architectures) to protect the client SLA during the investigation.

And one step to take **before** any tuning: capture one instance of the looping request (method, path, headers, which hop answers). One hour of work that turns a suspicion into hard evidence.

Does purging the stuck objects clear the storm?

Yes, mechanically. The storm's fuel is versions that exist in the source namespace: the scanner cannot re-probe what is no longer there. A full purge of an affected object, meaning **all of its versions and its delete markers**, removes them from every scanner walk, and their contribution to the probe rate drops to zero.

For data that ILM has already expired, this is semantically clean: the cluster stops paying to argue about objects that were scheduled to disappear anyway. It drains the current stock. It does not fix the code, and the next ILM purge cycle will build a new stuck population unless the exit condition is fixed.

One site or all sites? Deletion is the one operation that travels legitimately from source to replica, so purging on the source site propagates to its peers through the normal replication path. In a mesh that has already diverged, do not assume the propagation is complete: after the purge, verify on each peer that the copies are gone, and watch the 405 rate. A site that was offline or partitioned during the purge keeps its own copies and its own loop; repeat the purge there. Practically: purge the source, verify every peer, and treat "purge on all sites" as the fallback when verification shows survivors.

```
# one object: every version and every delete marker
mc rm --versions --force ALIAS/BUCKET/path/to/object

# a whole prefix whose content is expired anyway
# (destroys live data too: only for prefixes ILM was meant to empty)
mc rm --recursive --versions --force --dangerous ALIAS/BUCKET/expired-prefix/
```

Three precautions. The operation is destructive and irreversible: once the deletes replicate, the versions are gone on every site, so target only objects ILM already decided to expire, never a live prefix. It converts a 405 storm into a burst of legitimate delete replication, so batch it (a scripted loop with pauses, off peak) instead of wiping millions of versions in one pass. And verify with the same instrument that found the storm: the response-code distribution of the replication traffic should show the 405 share collapsing as the stuck population drains.

Keep the feature surface small

Run the smallest feature surface that satisfies the actual requirement. Look back at the four ingredients that had to stack for this storm to matter. Three of them are optional features: site replication, ILM version-purge rules, and a namespace that grew to tens of millions of versions under their combined watch. That is the pattern I keep seeing on MinIO platforms, and the advice I keep giving.

Every feature you switch on adds code paths that execute at scanner cadence across your whole namespace, and this post is what one of those paths does when its exit condition is wrong.

Concretely, on a large versioned MinIO namespace today:

- **Replication, bucket or site:** switch it on only if a second site is a hard business requirement, not an aspiration. It couples your sites through the scanner, its queue is blind and capped, and sync mode does not deliver the RPO its slides promise, as I detailed in [why sync mode will not save your RPO](#). A mirror is a trade, not a checkbox.
- **ILM tiering:** treat it as a second failure domain you must test, not a storage discount. A tiering defect I documented at the end of 2025 turned a plausible configuration into [silent data loss and fault-tolerance issues](#); if the data must leave the cluster, keep a local copy until a restore from the tier has been proven on your platform, not on a slide.
- **ILM expiration and noncurrent-version expiration:** legitimate on a single site, but on a replication mesh every purge-status marker becomes permanent replication traffic, as this storm shows. Phase them in, keep their scope narrow, and watch the replication traffic the day they deploy.
- **Community Edition since the licensing change:** features are not the only thing that shrinks the safety net; you also debug all of this yourself. Every optional feature you drop is surface you never have to diagnose at 2 a.m.

The stack that produced 684M useless requests in 48 hours was not bad luck. It was individually reasonable features stacked on top of each other: a mirror because DR, purges because storage cost, versioning because safety, all of it legitimate in isolation. A two-site mirror with no ILM purges and a moderate namespace would have produced nothing. Fewer features is not a lesser platform; it is the one that converges.

Lessons

The lessons are that a protocol using an error code as a success signal must close its loops, a replication catch-up path is a closed-loop system bounded by its drain rate, and error counters alone are not enough. The worst failure mode of this story passes for success in the metrics, and an unbounded backlog turns deterministic failures into permanent load.

- **A protocol that uses an error code as a success signal must close its loops.** The 405-signal here is elegant, but the success exit is gated by conditions that exclude precisely the population created by ILM, the component that triggers it.
- **A replication catch-up path is a closed-loop system.** Its work rate is $\min(\text{discovery}, \text{drain})$; if discovery is faster than the drain and deterministic failures never terminate, the load is proportional to the backlog, unbounded.
- **Error counters are not enough.** The worst failure mode of this story passes for success in the metrics. The response-code distribution, and a counter of versions in non-terminal status, are the bare minimum for a replication with a large namespace.

If you run MinIO in site replication with ILM rules and namespaces of tens of millions of versions: look at the response-code distribution of your replication traffic. A dominant 405 share has a very specific meaning. For the replication architecture side of the same platform, see my earlier note on [why sync mode will not save your RPO](#).

Analysis done on the MinIO open source code (archived master, AGPL). The mechanisms described (probe, gated exit, delete-on-replica guard) can be verified in `cmd/bucket-replication.go` and `cmd/erasure-object.go`. Platform figures are scaled and rounded to protect the client; ratios, orders of magnitude and mechanisms are unchanged.

FAQ

Is a 405 response on a MinIO replication request an error?

No. In MinIO site replication, a 405 on a versioned HEAD probe means the delete marker is already present on the target, so the replication is already done. The protocol deliberately reuses the S3 405 status code as an acknowledgment, and resync accounting counts it as a success.

Why do 405 responses saturate a MinIO cluster?

Delete markers that carry a version-purge status, created by ILM expiration flows, never reach the terminal Completed exit in `replicateDeleteToTarget`. Each scanner cycle re-probes the target, receives the deterministic 405, falls through to a `RemoveObject` that the replica guard blocks, and the version is re-queued. The probe rate is the scan rate multiplied by the stuck fraction, so the storm grows with the namespace instead of converging.

How do I detect this replication storm on my own cluster?

Measure the response-code distribution of the replication traffic, with `mc admin trace` filtered on the internal User-Agent or the replication frontend logs. A dominant 405 share, replication traffic at the same order of magnitude as client traffic, and a CPU load that carries no data are the signature. Also watch for versions stuck in a non-Completed replication status across scanner cycles. Native counters for neither exist today.

Does slowing the scanner fix the storm?

No. It divides the storm and the catch-up rate by the same factor, so it protects the client SLA while the investigation runs, but every stuck version still fails deterministically forever. The convergence path is a code fix that makes the 405 probe terminal for delete-marker operations, plus observability counters that surface the storm before it grows.

Which MinIO features should I avoid on a large namespace?

Treat every optional feature as a cost that scales with your namespace. The three that generated this storm and the earlier tiering incident are site replication, ILM tiering and ILM expiration rules: each runs extra code paths at scanner cadence across millions of versions. Keep the surface minimal: replication only when a second site is a hard requirement, expiration rules phased and narrow, tiering only with a proven restore path, and a documented rollback for each.

Can I stop the storm by deleting the stuck objects?

Yes, mechanically: a full purge of the affected objects (all versions plus their delete markers) with `mc rm --versions --force` removes them from the scanner's walk, so their share of the 405 traffic drops to zero. Purge on the source site first: deletion replicates legitimately to peers, but verify each site afterward and repeat the purge on any site that kept copies after a partition. Batch the deletes, expect a burst of legitimate replication traffic in their place, and treat the purge as draining the current stock, not as a fix: the code change is what stops the next ILM purge cycle from rebuilding the population.

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: Sync Mode Will Not Save Your RPO](#) - the replication architecture side of the same platform.

- [MinIO Tiering Warning: Data Loss and Fault Tolerance Issues](#) - why tiering needs a proven restore path, not a vendor slide.
- [MinIO and Lots of Small Files: Why It's Not a Database](#) - why object count and the scanner dominate operations.
- [MinIO on XFS: Inode Exhaustion and Prefix Design](#) - the filesystem substrate that sets scan-time reality.
- [Why File Count Matters as Much as File Size on Object Storage](#) - the namespace growth that feeds loops like this one.

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.