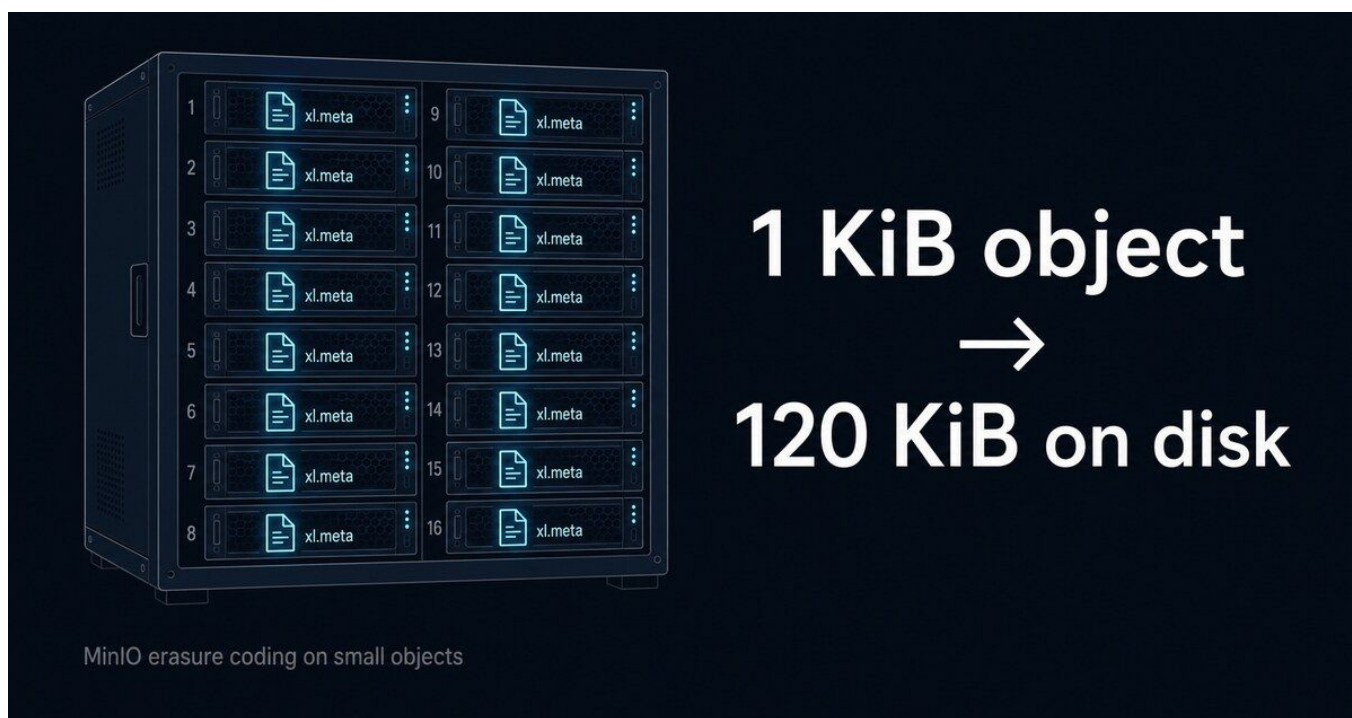


Stop using MinIO as a NoSQL database: why S3 object stores collapse on small-file workloads

Julien Laurenceau | Published 2026-05-17 | Updated 2026-10-01

<https://pepitedata.com/minio-not-a-nosql-database/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



TL;DR. In the last twelve months I have been called in on two independent MinIO platforms for a [data-platform performance audit](#), both suffering recurring outages and a hard scalability wall. Both clusters were healthy on paper: NVMe drives, modern hardware, recent MinIO release, plenty of free space. Both had been used, de facto, as a NoSQL key-value store: billions of objects in the kilobyte range. MinIO (and S3 in general) is not designed for that workload. This article explains why, with the numbers: LIST IOPS cost, XFS directory limits, scanner and heal duration, durability ceilings, and the storage-efficiency inversion that turns erasure coding into something *worse* than 3× replication for very small objects. It also explains why, regardless of workload shape, MinIO’s community edition is no longer a safe foundation for a new platform in 2026.

The pattern I keep seeing

The pattern I keep seeing is teams reaching for MinIO as a NoSQL database and failing in identical ways. S3-compatible storage has won: it is the default sink for data lakes, ML feature stores, observability pipelines, model registries, backup tools, and now AI artifacts. MinIO in particular is attractive: open source, on-prem, fast on NVMe, drop-in S3 API, so teams reach for it whenever they need “somewhere to put bytes addressed by a key”.

That last sentence is the trap. *Addressed by a key* sounds like a key-value store. S3 is not a key-value store. It is a **blob store with eventual list semantics**, designed in the early 2000s at Amazon for objects ranging from a few hundred kilobytes to multi-gigabyte files. Every benchmark MinIO publishes, every architecture decision in its codebase, every default tuning, assumes that shape of workload.

The two clusters I audited both drifted into a different shape:

- **Cluster A.** ML feature store. Each “feature vector” was stored as a separate S3 object, 200 to 800 bytes, addressed by <entity_id>/<feature_set>/<timestamp>. After 18 months in production: ~3 billion objects, mean object size 480 bytes, P99 PUT latency creeping from 30 ms to 4 s, daily timeouts on the consumer side, scanner running 11 days behind.
- **Cluster B.** Observability tier. Application emitted one S3 object per request trace, ~1.5 KB each, partitioned by service/YYYY/MM/DD/HH/. After 9 months: ~900 million objects, several leaf prefixes past 2 million entries, ListObjectsV2 on a single prefix taking 90+ seconds and saturating an entire EC set, triggering 503 SlowDown storms on concurrent writers.

Same root cause both times: **MinIO was being used as a NoSQL database**. Different team, different domain, identical failure mode.

What S3 (and MinIO) actually is, on disk

MinIO has no centralized metadata store. There is no equivalent of HBase’s META table, Cassandra’s system keyspace, or even a per-bucket index. The S3 namespace *is* the XFS directory tree itself: a bucket with 100 million small objects is 100 million directories on every drive of the erasure set, not a table with 100 million rows.

Concretely, for each S3 object, MinIO creates a **per-object subdirectory** on every drive of the erasure set, containing one x1.meta file (plus part.1, part.2... when the object is large enough to be sharded as separate files):

Example: object with key bucket/aa/bb/file.dat

```
/data/disk1/bucket/
├─ aa/                ← sub-prefix subdir (internal node)
│   └─ bb/            ← sub-prefix subdir (internal node)
│       └─ file.dat/   ← object subdir (leaf)
│           └─ x1.meta
```

The same tree exists on **every drive of the erasure set**. With a 16-drive set, 1 billion objects means 1 billion directories *per drive*, replicated 16 times across the set. There is no shortcut: every LIST, every scanner pass, every healing pass, every ILM cycle walks this directory tree.

This is the central fact you need to internalize before reading further. A bucket with 100 million small objects is not “a database table with 100 million rows”. It is **100 million directories on every drive of the erasure set**, and every operation on the namespace is a filesystem walk.

The LIST IOPS cliff

Listing is where MinIO’s per-object cost becomes a cliff: a cold ListObjectsV2 costs roughly 2 to 3 IOPS per object per drive, and a single flat LIST of 1 million objects can saturate an entire erasure set. Concurrent PUTs on the same set then see latency explode and time out with 503 SlowDown. Per-object cost of ListObjectsV2 on cold cache, per drive queried (XFS + MinIO):

Step	IOPS	Detail
readdir() of parent prefix	0.01 to 0.1 / object	Amortized over XFS dir blocks (~100 to 200 entries per 4 KB in leaf format;

Step	IOPS	Detail
		worse in B+tree format)
stat() / inode lookup of object dir	~1	Inode read if not cached
open() + read() of xl.meta	~1	Data block read
Total per object per drive (cold)	~2 to 3 IOPS	
Fully cached	0.1 to 0.5	readdir + metadata in RAM

This per-object cost is what makes flat LIST on a large bucket so dangerous. A single ListObjectsV2 walking 1 million objects on a 4-set, 60-drive cluster (EC:5 on 15), load distributed by hash, produces:

```

Per-drive cold IOPS  ≈ 2.5 × 1 000 000 / 15 ≈ 42 000 IOPS sustained
NVM budget          ≈ 50 000 IOPS / 4K random read
→ The set is saturated for the duration of the LIST.

```

A misplaced LIST holds the drives under sustained load for tens of seconds. Concurrent PUTs on the same set see latency explode and time out with HTTP 503 SlowDown. Clients that retry without backoff turn this into a retry storm: exactly what Cluster B experienced during its incident windows.

And the situation is *worse* when objects pile up in a single directory. XFS switches its directory layout from **leaf format** to **B+tree** somewhere around 200 000 entries. Beyond that threshold, sequential readdir becomes substantially more expensive, roughly 2× the per-entry cost, and the directory’s inode set no longer fits in the page cache. A 1-million-entry directory holds ~400 MB of xl.meta alone, which evicts under any concurrent I/O pressure.

XFS directory thresholds (the numbers I use)

The one number to remember is 10 000 entries per leaf prefix per drive. Past that you pay a tax on every LIST, every scanner pass and every heal; past 50 000 you are inside MinIO’s own alert zone; past 200 000 XFS switches to B+tree and the cost model has shifted under you.

Entries per directory (per drive)	Behavior
< 10 000	Nominal readdir, fits XFS leaf format
10 000 to 50 000	Acceptable; readdir slowdown begins to be measurable
50 000 to 200 000	Significant readdir cost; MinIO scanner alert at 50K (scannerExcessFolders)
~200 000+	XFS switches to B+tree: sequential readdir becomes substantially more expensive

If you remember one number from this article, remember **10 000 entries per leaf prefix per drive**. Past that, you are paying a tax on every LIST, every scanner pass, every heal. Past 50 000 you are inside MinIO’s own alert zone. Past 200 000 you are in XFS B+tree territory and the cost model has shifted under you.

Object count: what is actually safe per bucket

Safe is under 100M objects per bucket with well distributed prefixes: that is where scan, ILM and listings remain usable in hours to days. MinIO does not document a hard ceiling, and the practical limits are real and structural: 100M to 500M brings progressive degradation of LIST, scanner and ILM, above 500M

maintenance ops become problematic, and past 1B MinIO is not designed for the case.

Object count per bucket	Behavior
< 100M	Nominal performance if prefixes are well distributed
100M to 500M	Progressive degradation of LIST, scanner, and ILM. Full scan can take multiple days
> 500M	Risk zone. Maintenance ops (heal, ILM, list) become problematic
> 1B	Not recommended. MinIO is not designed for this case

Why “< 100M” when 655M is theoretically reachable? A perfect 2-level hash partition ($256 \times 256 \times 10\,000 = 655M$) makes the scanner walk 65 536 leaf directories of 10 000 entries each. The dominant cost is per-object metadata work (~ 2.5 IOPS cold), not the inter-folder sleep. The scanner needs **16 cycles** (`dataUsageUpdateDirCycles`) to cover the namespace under hash sampling. Full coverage drifts into **weeks**. The 100M ceiling is where scan, ILM and listings remain usable in **hours to days**, not weeks.

And remember: **versioning multiplies this**. With N versions per object on average, the on-disk file count is $N \times$ the logical count. **Site replication requires versioning**, so every multi-site MinIO deployment is silently running with this amplifier in force, unless a strict non-current-version ILM policy clamps it.

Durability and healing: the silent cost

The silent cost is a healing SLA measured in weeks on a system you provisioned for one measured in hours, and it stays invisible until something fails. MinIO’s heal path is gated by `healObjectSelectProb = 1024`, so only 1 object in 1024 is checked per scanner cycle. On a LOSF cluster the scanner takes 10 to 14 days to complete, so a corrupted shard can sit untouched for two weeks.

MinIO’s heal path is gated by `healObjectSelectProb = 1024`: only **1 object in 1024 is checked per scanner cycle**. On a billion-object namespace that is ~ 1 million heal checks per cycle, which sounds manageable. The problem is the cycle duration. On a LOSF cluster where the scanner takes 10 to 14 days to complete, **a corrupted shard discovered on day 1 may not be touched by the heal path for two weeks**. If a second drive fails in that interval, you may discover that an object you thought was safe under EC:4 parity is actually unrecoverable.

The MinIO defaults assume the scanner completes within hours. On LOSF at scale, it does not. You are silently operating with a **healing SLA measured in weeks**, on a system you provisioned for a healing SLA measured in hours.

Two practical knobs make this worse, not better:

- `bitrotscan=on` is a major LOSF amplifier. Default is off. Enabling it makes the scanner read the *data shards* of every object during the sweep, not just `x1.meta`. On LOSF this multiplies scanner I/O by orders of magnitude. Past a few hundred million objects, it is typically not viable.
- The OSS scanner **never aborts**. At 50K subdirectories per node it emits an alert; at 250K it forces stats-tree compaction; it continues scanning indefinitely. There is no break, skip, or abandon path in the OSS code. The AIStor (commercial) build adds an explicit skip at 5M subdirectories per prefix. If you are running OSS at scale and you have such a prefix, the scanner is stuck on it every cycle.

The storage-efficiency inversion (the part nobody talks about)

The inversion is concrete: for 1 KiB objects, EC 8+8 on a 16-drive set consumes 8 to 10× more raw disk than 3× replication on a system designed for small values. This is the most counter-intuitive consequence of LOSF on MinIO, and the one I had to derive carefully on Cluster A before presenting it to the CTO.

Erasure coding is sold as the storage-efficient alternative to replication. EC 8+8 on a 16-drive set has a logical overhead of 2× (16 shards for 8 data shards), versus 3× for triple replication. On large objects, this holds: a 1 GiB object stored on EC 8+8 consumes ~2 GiB of raw disk vs 3 GiB under 3× replication.

For small objects, the relationship inverts. Here is why.

Mechanism. When an object is below 256 KiB (default `storageclass.inline_block_size`), MinIO inlines the object's payload inside `xl.meta` itself, rather than writing separate `part.N` files. The payload is still erasure-coded: each drive's `xl.meta` contains that drive's shard inline, plus the message-pack envelope, checksums, version metadata, and shard layout. There is one `xl.meta` per drive of the erasure set, each in its own per-object subdirectory.

Three sources of inflation that dominate the data size for small objects:

- 1. Per-drive metadata envelope.** The `xl.meta` msgpack header carries object metadata, ETag, version ID, modtime, checksum block, EC layout, content-type, user metadata. On a tiny object this envelope alone is several hundred bytes, already comparable to or larger than the payload itself. And it is duplicated on every drive.
- 2. Filesystem block granularity.** XFS allocates data in 4 KiB blocks (default). A 200-byte `xl.meta` consumes one 4 KiB data block on disk. With a 16-drive set, that is **16 × 4 KiB = 64 KiB of allocated data blocks per logical object**, regardless of how small the payload is.
- 3. Directory and inode overhead.** Each per-object subdirectory itself consumes an inode and a directory data block on every drive. Add that × 16.

Concrete arithmetic, 1 KiB logical object on a 16-drive EC 8+8 set:

Layer	Per drive	× 16 drives
Inlined shard (1 KiB / 8 data shards ≈ 128 B)	~128 B	2 KiB raw shards
<code>xl.meta</code> envelope (msgpack + metadata + checksums)	~400 to 800 B	~6 to 12 KiB
<code>xl.meta</code> rounded to XFS 4 KiB data block	4 KiB	64 KiB
Per-object subdir inode + dirent block	~4 KiB	~64 KiB
Effective on-disk		~120 KiB for a 1 KiB object

Compare to a 3× replicated 1 KiB object on a sane store (Ceph RGW small-object handling, or a key-value engine): ~3 × 4 KiB block + 3 inodes ≈ **12 to 15 KiB**.

The storage efficiency ratio inverts by roughly an order of magnitude. EC 8+8 on a 16-drive set, for 1 KiB objects, consumes 8 to 10× more raw disk than 3× replication on a system designed for small values. The larger the erasure set, the worse it gets: a wider EC set means more drives, more `xl.meta` copies, more 4 KiB-block roundings, more per-object inodes.

To be precise on the mechanism: the inlined data is not literally copied 16 times. It is sharded by the EC encoder and each drive holds its own shard inside its own `x1.meta`. The amplification is not from data duplication: it is from **the per-drive metadata envelope plus the filesystem's minimum block allocation, multiplied by the erasure set width**. For small enough objects, those fixed overheads dwarf the shard payload, and the system behaves *as if* it were replicating the object across all drives of the set. The user-facing storage efficiency curve crosses below 3× replication somewhere around the **tens-of-kilobytes** range, depending on EC width and FS block size, and gets dramatically worse below 4 KiB.

This is why pricing capacity for a LOSF MinIO cluster using “EC 8+8 = 2× overhead”, the figure on every MinIO sizing slide, produces estimates that are wrong by **3 to 10×** on the actual disk footprint. Both clusters I audited had been sized this way. Both ran out of disk far earlier than the procurement plan expected.

A note on the MinIO project itself

Independent of the LOSF question, MinIO's open-source posture has changed in a way that matters for any team picking it for a new platform in 2026. The server is AGPL-3.0 rather than Apache, which rules out embedding it in a product for most enterprises, and the community edition is effectively in maintenance mode while new capabilities ship in the commercial **AIStor**. Pricing that license into the TCO is now part of any MinIO decision.

- **License.** MinIO server is **AGPL-3.0**, not Apache. For most enterprises this rules out embedding it in a product and creates legal friction around any internal modification or fork. It is not equivalent to the Apache-licensed S3-compatible options (Ceph RGW, SeaweedFS) on procurement and compliance grounds.
- **Community edition is effectively in maintenance mode.** The embedded web console was removed from the OSS build after RELEASE.2025-04-22T22-12-26Z. Several admin features have followed the same path. New capabilities ship in the commercial **AIStor** product, not in the CE codebase. A serious production deployment in 2026 means buying AIStor, at which point the comparison is against commercial Ceph (Red Hat / IBM Storage Ceph), commercial SeaweedFS support, or directly against AWS S3 / GCS / Azure Blob, not against “free MinIO”.

I am not telling anyone to rip out an existing MinIO cluster. I am saying: **do not start a new platform on community-edition MinIO in 2026 without pricing the AIStor license into the TCO from day one**. And whichever edition you run, none of it fixes the LOSF problem above: that one is structural.

What to do instead, if your workload looks like a NoSQL store

If you are reaching for S3 because you want “a place to put a key and get bytes back, addressed by name, accessed at high concurrency, with billions of small entries”, you do not want an object store. You want a database.

Apache Cassandra: the default answer for write-heavy small-payload workloads

For any workload that looks like **billions of small writes, payload below ~500 KiB, addressed by key, with predictable access patterns**, Apache Cassandra is the right tool, and on most criteria it is the *strictly better* tool than MinIO, not just a different one.

Concretely, what Cassandra gives you on this workload that MinIO does not:

- **Write-optimized storage engine.** Cassandra's LSM-tree (memtable → SSTable) is purpose-built for high write throughput. Writes hit the commit log and an in-memory memtable, flushed in batches as immutable SSTables: no per-object subdirectory, no per-write readdir, no XFS B+tree blow-up. The whole class of failures described above simply does not exist.
- **Compression out of the box.** SSTables are compressed by default (LZ4, with Zstd / Snappy / Deflate available), block-level, with per-block checksums. On compressible small payloads (JSON, traces, feature vectors, logs) the on-disk footprint is typically **3 to 5x smaller than the logical data**: the opposite direction of MinIO's small-object inflation. Combine this with the storage-efficiency inversion above and you can see clusters where switching the small-record path from MinIO to Cassandra divides total disk consumption by 20 to 30x.
- **Real durability semantics, not "trust the scanner".** Every write goes to a commit log (fsync-able, configurable per table) before acknowledgement. Hinted handoff covers brief node outages. Anti-entropy repair (nodetool repair, incremental or full) is a first-class operation with a documented SLA, not a 14-day silent scanner job gated by a 1/1024 probability constant. Read-repair runs inline on every read. The end-to-end durability story is *engineered*, not emergent.
- **Tunable consistency, per query.** ONE, QUORUM, LOCAL_QUORUM, EACH_QUORUM, ALL. You choose per statement whether you want speed, intra-DC quorum, or cross-DC quorum. MinIO has one knob: the EC stripe inside a single site. There is no equivalent of LOCAL_QUORUM or EACH_QUORUM across MinIO sites. For workloads with a regulatory or financial durability requirement, this alone disqualifies MinIO.
- **Real multi-DC replication, with conflict resolution that is at least defined.** Cassandra's NetworkTopologyStrategy lets you declare a replication factor per datacenter (e.g. {DC1: 3, DC2: 3, DC3: 2}) and choose the consistency level per query. Writes can require acknowledgement from one or more remote DCs *before returning success*. Cells carry vector-style timestamps and conflicts resolve at the cell granularity (last-write-wins on cell timestamp, with all the well-understood caveats around clock skew). It is not magic: LWW is still LWW, but the building blocks are exposed and the topology is explicit.

Contrast with MinIO "site replication". It is worth being precise here, because the marketing language obscures what the feature actually is.

MinIO's `mc admin replicate` is an **asynchronous mirror**, not a distributed-consensus replication layer. Each site is a fully autonomous cluster. There is **no quorum across sites**, no consistency level that requires acknowledgement from a peer site before the local PUT returns 200. A network partition that splits a 3-site mesh into {A} and {B, C} lets both sides continue accepting writes: there is no minority-side fencing. On heal, conflicts on the same object key are reconciled by **last-write-wins on the object timestamp**, with no application-visible conflict signal, no version vector, no per-attribute merge, no escalation path. If clocks drift between sites (NTP failure, VM pause, leap second), the LWW outcome can silently discard the "correct" write. The replication is closer to a continuous rsync than to Cassandra's NetworkTopologyStrategy: useful as a DR mechanism, dangerous if you mistake it for a multi-master database.

For any workload where two sites might write to the same key, or where a regulator will ask "what guarantees does this system make in the event of a network partition", MinIO site replication is not the answer. Cassandra, configured with the right replication factor and consistency level, is.

Side-note on MinIO --sync replication: don't. Whatever you do, do not enable synchronous mode (`mc admin replicate update --mode sync`, or `--sync` on a bucket-replication rule) thinking it gives you cross-site write quorum. It does not. In sync mode, MinIO still does **not** fail the source PUT if the remote site is unreachable: the local write succeeds, the object is marked PENDING, and the scanner replays it when the remote returns. What you actually get is: every PUT now pays the round-trip latency to every peer site on the happy path, with **no additional durability guarantee** when it matters

(during a partition, the only moment a synchronous protocol would justify the latency cost). Throughput collapses, tail latency explodes, and the failure mode is unchanged from async. On top of that, mixing per-bucket --sync rules with a site-replication-managed bucket is an unsupported configuration that the system does not enforce: SR will silently overwrite your manual rule on peer add/edit/resync/upgrade, and `mc admin replicate status` will report the bucket as out-of-sync even when replication is working. If you genuinely need synchronous cross-site semantics, you need a different storage engine: not a flag on MinIO.

Sizing rule of thumb. Cassandra handles cell values up to a few MiB without trouble. Anywhere between **a few hundred bytes and ~500 KiB per payload** is the sweet spot. Above ~1 MiB per value, partition size and compaction cost climb non-linearly and you should split the payload (chunking) or offload the blob to an object store and keep only the pointer in Cassandra. Which brings us to:

The hybrid pattern that almost always wins

Cassandra for the small-record path (metadata, traces, features, registry entries, references), object store for the actual blobs (training files, model artifacts, raw events batched into Parquet). The Cassandra row holds the blob's S3 key, ETag, size and any application metadata; the object store holds the bytes. Reads are key-direct on Cassandra and a single GET on the blob store. There is no LIST. Ever.

This is the architecture every team I have audited ended up moving toward. Starting there from day one is much cheaper than rebuilding.

When Postgres is enough

If your scale is below ~100M rows and you do not need horizontal scaling, a plain Postgres table with a bytea column is a perfectly serious answer. Yes, really. You get ACID transactions, mature backup tooling, point-in-time recovery, well-understood operational behavior, and a query language. Most teams reach for an object store at this scale out of habit, not because they have measured Postgres and found it wanting. Measure first.

If you genuinely need an on-prem S3 API

For the blob side of the hybrid pattern, or for any other workload where an S3 API is a hard requirement, my 2026 recommendation is short:

- **Ceph (RGW)** for any production platform where durability matters and silent data loss is not an option. Mature, properly licensed (LGPL), well-instrumented, scrub and deep-scrub are first-class, commercial support available from multiple vendors (IBM/Red Hat, Clyso, 42on, Croit). The operational learning curve is real, but it is the only on-prem S3 I would put critical data on in 2026. Same caveat on small objects: RGW is not magic on LOSF either, and small-object workloads still need the “don’t use it as a NoSQL store” discipline.
- **SeaweedFS** (Apache 2.0) for less critical projects, or when the dominant workload is small files and a different on-disk layout helps (volume-based, with a metadata index designed for LOSF). Lighter to operate than Ceph; correspondingly less battle-tested at multi-petabyte scale and on long-tail failure modes.

MinIO does not appear on this list. Given the AGPL license, the CE-vs-AIStor split, and the structural LOSF behavior documented above, I do not currently recommend starting a new on-prem S3 platform on it in 2026.

How to recognize you are heading for this wall

You are heading for this wall when mean object size drops below 64 KiB and keeps falling, LIST p95 latency creeps up week-over-week, or the scanner cycle takes more than a day. Two more signals complete the set: PUT 503 SlowDown responses correlated with LIST traffic, and disk usage diverging from the sizing model. Five signals from the two audits, in order of how early they showed up:

1. **Mean object size is below 64 KiB and trending down.** Check `mc du` or scrape Prometheus `minio_bucket_usage_object_total / minio_bucket_usage_total_bytes`.
2. **LIST p95 latency is creeping up week-over-week.** Not “is it fast” but “is it getting slower than last month”.
3. **Scanner cycle duration is climbing.** Visible in `mc admin info` or in the scanner’s own audit events. Anything past 24h is a yellow flag; past 7 days is a red flag.
4. **PUT 503 SlowDown responses correlated with LIST traffic.** This is the diagnostic for the IOPS-burning LIST scenario. `mc admin trace --call s3.ListObjectsV2` on the server side will tell you which LIST is doing it.
5. **Disk used / disk projected is diverging from the sizing model.** This is the storage-efficiency inversion above, showing up in capacity planning.

If you see two or more of these, you are not in an operational incident yet: you are in a design problem that will eventually express itself as one. The fix is not “tune the scanner”, it is **move the small-file path off the object store**.

If you are already in this situation

You have three options: compacting small writes at the source, carving the small-record hot path out to another store, or re-shaping the namespace so every leaf prefix stays under 10 000 entries per drive. Option (1) and (2) are the only ones that scale, and option (3) is a holding action. In increasing order of cost:

1. **Compact at the source.** Batch the small writes upstream into larger objects (Parquet, Avro, line-delimited JSON, tar). Most LOSF problems originate from one or two writers that emit one object per record. Find them and aggregate.
2. **Carve out the hot path to a different store.** Move the small-record workload to Cassandra (or Postgres, if the scale fits), leave the cold/historical/blob data on the existing object store. This is the architecture the team probably should have started with.
3. **Re-shape the namespace.** If neither of the above is possible, redesign the prefix scheme to keep every leaf prefix under 10 000 entries per drive, and enforce a non-current-version ILM policy if versioning is on. This buys time. It does not fix the storage-efficiency inversion or the scan-duration problem.

Option (1) and (2) are the only ones that scale. Option (3) is a holding action.

Support

If your MinIO cluster is exhibiting any of these symptoms, an independent expert review is usually faster and cheaper than another sprint of internal investigation. [Book a 30-minute scoping call](#) or an [Expert Call](#) if you want one focused hour on the question.

Update 2026-09-29. This post is now part of [Object Storage for Data Platforms at Scale](#), the field guide to running object storage at scale.

Related posts

- [Object Storage for Data Platforms at Scale](#)
- [MinIO and lots of small files: why it's not a database](#)
- [MinIO on XFS: inode exhaustion and prefix design](#)
- [Why file count matters as much as file size on object storage](#)

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