

MinIO on XFS: Inode Exhaustion and Prefix Design for Lots of Small Files

Julien Laurenceau | Published 2026-07-27 | Updated 2026-10-21

<https://pepitedata.com/minio-xfs-inode-exhaustion-prefix-partitioning-losf/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau

TL;DR. MinIO does not keep a global object index in RAM. On directly attached drives it maps each S3 key onto a real XFS path and stores per-object metadata in `xl.meta`, written across every drive of the selected erasure set. On Lots of Small Files (LOSf), free TiB keeps looking healthy while free inodes and directory-walk cost collapse. This post covers three things I keep re-teaching on audits: how MinIO actually uses XFS, why inode exhaustion is a first-class capacity metric, and how to design S3 prefixes so leaf directories stay workable. It is a companion to [MinIO and lots of small files](#) and [when erasure coding becomes 15× replication](#).

The failure mode you will misread as "storage is fine"

The failure mode you will misread as "storage is fine" is inode and metadata IOPS exhaustion: free TiB keeps looking healthy while PUTs time out, a hot LIST saturates one erasure set, and the scanner falls days behind. A typical LOSf shape on bare metal looks like this:

- most objects in the **few-KiB to tens-of-KiB** band (classic small business objects, features, events, document units),
- sustained ingest on the order of **hundreds to a few thousand objects per second** in the busy window,
- retention long enough that the active namespace can reach **hundreds of millions to billions** of logical keys,
- fixed drive count (no elastic multi-tenant S3 backend behind you).

Ops dashboards then show green capacity. `df -h` still has TiB free. PUTs start timing out. LIST of a hot prefix saturates one erasure set. The scanner that feeds heal selection and lifecycle work falls days behind. Notification spools fill by *file count*, not by bytes. You are not out of disk in the usual sense. You have run into the filesystem cost of "one small S3 object equals real metadata work on N drives".

If you remember one framing: on a LOSf MinIO cluster you are buying **inodes and metadata IOPS**, not mostly disk bytes.

How MinIO uses XFS (the on-disk truth)

MinIO uses XFS to map the flat S3 keyspace onto real directories: on local drives, each `/` in an object key becomes a real directory. Recommended practice is XFS on directly attached JBOD-style devices; that is the substrate MinIO's guidance assumes for large deployments, and XFS is not a MinIO database. It supplies directories, files, inodes, allocation, caching, and durability semantics while MinIO does the object layout and the erasure math on top.

S3 key → directories → `xl.meta`

Illustrative layout under one drive's data path:

```

/data/minio/<bucket>/
└─ aa/                # prefix segment (real directory)
   └─ bb/             # deeper prefix segment
      └─ object-id/    # per-object directory (leaf)
         └─ xl.meta     # MessagePack metadata (+ inlined payload when small)

```

Facts that matter more than the picture:

- **No global in-RAM object index for ordinary placement.** Metadata lives on disk in `xl.meta`. Process RSS tracks concurrent work, not total object count. That is how MinIO escapes the classic HDFS NameNode heap ceiling. It does not mean metadata became free.
- **Per erasure set, not once cluster-wide.** MinIO hashes the full object name, picks one erasure set, and writes object material onto every drive of that set. Prefix hashing is *not* required merely to spread writes across sets; the set selection already uses the full key.
- **Inlining of small shards.** Current MinIO storage-class logic decides inlining from *per-drive shard size*, with a default `inline_block` on the order of **128 KiB** (and a stricter effective threshold on versioned buckets). When inlined, payload lives inside `xl.meta`. A small PUT is still N filesystem create/write paths on an N-drive set, not one tiny file in a shared pool.
- **Non-inlined objects add more files.** A single-part object typically adds a UUID data directory and `part.1` next to `xl.meta`. Multipart grows that further. Shared prefix directories, multipart upload staging, trash, notification spools, and `.minio.sys` all add inodes on top of the pure object model.
- **Page cache is the soft metadata cache.** It is shared with data, not guaranteed, and disappears under pressure. Treating page-cache growth as "wasted RAM" on a LOSF platform is a common miss-read.

Inode amplification (what one object costs)

Let O be distinct object keys, D total drives in the pool, N drives per erasure set, and $S = D/N$ sets.

- For a typical **inlined** object: each selected-set drive holds the object directory plus `xl.meta`, about **2 inodes**. Expected per-drive use is about $2 \times O / S$, equivalently $2 \times O \times N / D$.
- For a typical **non-inlined single-part** object: add the data directory and `part.1`, about **4 inodes** per selected-set drive.
- **Parity does not reduce metadata copies.** Parity changes usable capacity and fault tolerance. Set width N is what multiplies metadata. With fixed D , a larger N means fewer sets and more object metadata sitting on each drive.

Worked order of magnitude: **100 000 objects under one application prefix** with $N = 15$ is not "100k tiny files". Across the set you are looking at roughly **1.5 million** filesystem entries for the inlined case alone (object dirs + `xl.meta`), plus parent prefixes. That is why small mean object size makes free-byte charts look safe for a long time.

Two different directory ceilings (files vs subdirs)

A single XFS directory holds different kinds of entries. Treat them separately in design:

- **Object leaves** (many object directories and their `xl.meta` under one final prefix): keep well under a house target of about **10 000 objects per leaf**. Past that, `readdir`, `ListObjects*`, concurrent PUT/LIST, and scanner leaf walks get expensive. Severe pain often shows well before a theoretical "maximum".
- **Prefix fan-out** (many child directories under one parent): MinIO's scanner documents an **excess subfolders** guardrail. Public docs describe an alert past about **50 000** subfolders under a prefix, with

higher error and skip thresholds beyond that. That is a scanner operational limit, not "XFS cannot store more entries".

Design rule that holds up in the field: **internal nodes hold only subdirectories; leaves hold only objects**. Do not dump a huge flat object pile and a huge child-prefix fan-out into the same directory.

XFS internally progresses through short-form, block, leaf, and node directory formats; large directories end up as B+trees. Point lookup can stay decent. A full `readdir` still scales with entry count and the metadata working set. Transitions depend on block size, name length, and layout, so I do not publish a single universal "at 200 000 entries Y happens" number. Benchmark your leaf sizes. Use the MinIO scanner thresholds for hierarchy depth decisions.

Inode exhaustion: mechanics and field signals

Inode exhaustion is a first-class capacity metric on LOSF MinIO: free bytes stay comfortable while free inodes collapse, since each logical object creates multiple inodes and several small writes on the selected erasure set. XFS allocates inodes dynamically, but `mkfs.xfs maxpct` caps how much of the filesystem may be spent on inode blocks, so LOSF workloads must plan inode count explicitly. Treat inode `ENOSPC` as a distinct failure class from byte-capacity exhaustion.

Why free space lies

On LOSF with inlining, average useful payload may be a few kilobytes while each logical object still creates multiple inodes and several small writes on the selected set. Free **bytes** stay comfortable while free **inodes** collapse.

XFS allocates inodes dynamically. Dynamic is not unlimited. `mkfs.xfs's maxpct` caps the percentage of the filesystem that may be spent on inode blocks (documented defaults step down as the volume grows: on the order of 25% below 1 TiB, 5% below 50 TiB, 1% above 50 TiB). You can raise it later with `xfs_growfs -m`, at the cost of data capacity headroom. XFS can also report out of space while free blocks remain if fragmentation leaves no suitable extent for another inode chunk; sparse-inode support mitigates this, it does not remove the capacity planning problem. Quotas and allocation-group locality can bite separately.

Default `mkfs` inode density is sized around ordinary byte-capacity workloads. For "one inode per a few hundred bytes of average logical object after EC amplification", you must plan **inode count explicitly**.

Amplifiers that look optional and are not free

- **Erasureset width N** - roughly linear metadata multiplication per object.
- **Versioning** (often forced by site replication) - versions share the object directory and the same `xl.meta`. Inline versions and delete markers are records inside that file; they do *not* each create a fresh `xl.meta` inode. Non-inline versions usually add data directories and parts. Still, versioning grows `xl.meta`, scanner work, and often delete-marker clutter. Measure versions per object. Do not assume 1.0 forever. Unbounded noncurrent cleanup debt has shown up as sudden compaction-class surprises, not as a steady linear tax only.
- **Local notification spools** - store-and-forward on disk can be another LOSF tree. Watch space *and* inodes on that mount.
- **Unbounded batch extract / snowball-style fan-out** - one API call creating 10^2 - 10^6 objects is normal on elastic multi-tenant cloud S3 and brutal on a fixed 60-drive cluster.
- **Flat or sequential keys** - silent creation of monster leaf directories.

What pressure looks like operationally

- `df -i` on every data mount trending down while `df -h` is still green,
- create / PUT failures or latency cliffs with "no space" class errors despite free bytes,
- `ls/find` on hot data paths becoming operationally dangerous,
- notification spool directories dirty on file count,
- heal, ILM, and replication retry looking "stuck" because they share the same walk physics,
- object count rising much faster than used bytes (mean object size collapse = LOSF signature).

Treat inode `ENOSPC` as a distinct failure class from byte-capacity exhaustion. Raising `maxpct` is mitigation and headroom trade, not a substitute for controlling object creation rate.

Why background work turns LOSF into a durability problem

Background work turns LOSF into a durability problem because MinIO background progress is largely **scanner-driven**: data-usage stats, bitrot sampling and heal selection, lifecycle enqueue, and some replay paths for replication failure tails. There is no separate ILM engine that enjoys a free $O(1)$ index of "everything that expires tomorrow". On LOSF the scanner falls days or weeks behind, and every safety net falls with it.

On a healthy medium-object cluster, a full pass is background noise. On LOSF blueprints, full coverage drifts from hours toward **days or weeks** once you sit in the tens or hundreds of millions of objects. Then you get silent erosion of safety nets in steady state:

- bitrot / heal discovery lag,
- ILM expiry lag (retention and compliance drift),
- replication PENDING/FAILED tails that only re-enter after a slow sweep,
- metrics that describe the last completed walk, not instantaneous truth.

Prefix partitioning and careful ILM slow the collapse. They do not remove the structural cost of one filesystem object per business unit at extreme ingest rates.

Tag updates rewrite the object

Tag updates rewrite the object: tags look like metadata, so teams assume updating them is cheap, but on inlined small objects that assumption is expensive. The cost of tagging scales inversely with object size: the operation that is free at 10 MiB is close to a full rewrite at 20 KiB, which is exactly the band LOSF platforms live in. It is the same inversion the rest of this post keeps producing.

Tags live inside `xl.meta`. There is no in-place patch for a field inside that file. A `PutObjectTagging` reads `xl.meta`, modifies the version's metadata, and writes the whole file back, on **every drive of the erasure set**, under the same write quorum a PUT needs. On a versioned bucket it does not create a new version, which is the one piece of good news here.

Now apply inlining. For a small object the payload *is* inside `xl.meta`. Rewriting the metadata rewrites the payload with it. So a tag update on a 20 KiB inlined object touches the same file count on the same drives and moves broadly the same bytes as the original write. You skip the erasure compute, which at this object size is noise next to the metadata write path. You do not skip the write path itself.

Contrast a multi-MiB object: `x1.meta` is small and sits next to `part.1` rather than containing it, so a tag update really is a cheap metadata operation. **The cost of tagging scales inversely with object size.** The operation that is free at 10 MiB is close to a full rewrite at 20 KiB, which is exactly the band LOSF platforms live in.

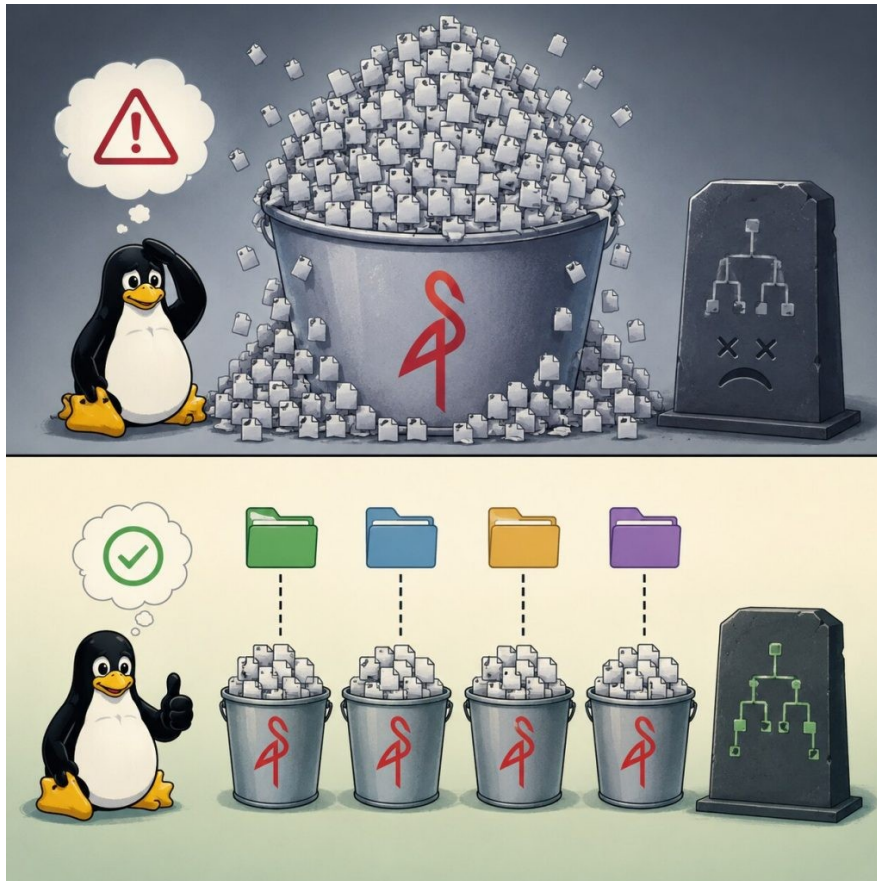
Where this bites in the field:

- **Tag-driven ILM.** Filtering lifecycle rules on tags is idiomatic S3 and I have no argument with the pattern itself. What it invites is the bulk retag campaign. Retagging 100 million small objects to trigger a transition or an expiry is a full rewrite of that namespace, at PUT cost, running on top of live ingest and competing for the same drives.
- **Classification after ingest.** Pipelines that land the object first and tag it once a downstream job has decided something about it have quietly doubled the write cost per object. Capacity plans in this shape are almost always sized for one write.
- **Tags used as mutable application state.** Tags are not a key-value store bolted onto the object, and treating them as one on LOSF is the same failure as [using MinIO as a NoSQL database](#), arriving through a different door.
- **Versioning compounds it.** `x1.meta` carries the records for every version of the object, so it grows over the object's life. The rewrite grows with it, and each tag update also fires a notification event when you have one configured.

The rule I hand producers: **decide tags at PUT time.** If a tag has to change after ingest, budget it as a write, not as metadata. If you need genuinely mutable per-object attributes at LOSF scale, keep them in a database and keep the object immutable. Prefix design does nothing for you here. This one is a write amplifier, not a directory problem.

Recommended prefix partitioning for LOSF

Recommended prefix partitioning for LOSF keeps leaves bounded: under about 10 000 objects per leaf and 2 to 4 useful key depth levels. The important negative result is that **prefix partitioning does not materially reduce total per-object inode use.** It adds a few directory inodes, but it stops millions of object directories from landing in one physical XFS directory and gives clients and MinIO bounded subtrees to list and scan.



Both halves hold the same objects and cost the same inodes. Partitioning changes how many object directories land in one physical XFS directory, and that is what `readdir`, `LIST`, and the scanner actually pay for.

Targets I use in practice (heuristics, not product hard limits)

- **< ~10 000 objects per leaf directory** as a conservative house target; alarm well before tens of thousands inland if latency or scanner lag appears earlier.
- **< ~50 000 child prefixes per directory**, aligning with MinIO's scanner excess-subfolder alert region; go deeper instead of wider when you approach it.
- **2-4 useful key depth levels** for operational topology. Too shallow builds monsters. Too deep adds path cost and human pain.
- **Active objects per bucket** large enough to make day-scale full walks painful (order-of-magnitude "tens to low hundreds of millions", workload-dependent) should force automation and often data-model changes, not only prettier keys.

Capacity mental model:

$$\text{max_objects} \approx (\text{fanout} \wedge \text{depth}) \times \text{objects_per_leaf}$$

Run that formula across the schemes producers actually propose, holding the leaf at 10 000 objects, and the shape of the trade-off becomes obvious. The last column is the one people forget: a scheme can be perfectly sized at the leaf and still put too many children under one parent.

Depth	Key scheme	Fan-out per level	Theoretical ceiling	Widest directory vs the 50k alert
0	bucket/obj	flat	10 000	the leaf itself is the problem

Depth	Key scheme	Fan-out per level	Theoretical ceiling	Widest directory vs the 50k alert
1	{hex2}/obj	256	2.5M	256, comfortable
2	{hex2}/{hex2}/obj	256 × 256	655M	256, comfortable
2	{YYYY}/{MM-DD}/obj	1 × 365	3.6M per year	365, comfortable
3	{hex2}/{hex2}/{hex2}/obj	256 ³	~167B	256, comfortable
3	{YYYY}/{MM}/{DD}/obj	1 × 12 × 31	3.7M per year	31, comfortable
4	{YYYY}/{MM}/{DD}/{HH}/obj	1 × 12 × 31 × 24	89M per year	24, comfortable

Theoretical bucket ceiling assuming 10 000 objects per leaf and even distribution. Ceilings, not targets.

Two things to read out of it. First, **fan-out beats depth** for raw capacity: one extra hex level multiplies the ceiling by 256, while one extra calendar level multiplies it by 12 or 31. Pure time keys hit a per-year ceiling that no amount of depth fixes, which is why the recommended layout below puts a hash *under* the time levels rather than choosing between them. Second, the numbers assume even distribution. Real producers are skewed, so treat the ceiling as the best case you will never reach and size for your hot leaf, not your mean leaf.

None of this is permission to stop thinking. A 2-level hex split gives hundreds of millions of theoretical keys, and walking tens of thousands of hot leaves with multi-IOPS cold meta work per object still turns one scan cycle into days. The ceiling is a storage arithmetic limit. The operable limit arrives much earlier and is set by scan time, ILM lag, and LIST p99 on *your* hardware class. Benchmark those.

Sizing leaves under a hash fan-out

If K unique keys arrive in a time window and you use H hash buckets under that window, expected leaf packing on a given drive is roughly $K / (H \times S)$ after set selection (the full key hash already picks the set; the extra hash prefix exists for directory balance inside application-visible trees). Choose H so the expected leaf stays under your target with skew and growth headroom.

Example arithmetic I use in workshops (orders of magnitude only):

- ~1 800 obj/s × 600 s (10-minute tranche) ≈ **1.08M** objects if you only time-bucket → far above 10k/leaf.
- Add a two-hex hash (×256) under the tranche → about **4.2k** objects per leaf per tranche → under the house rule while peak load is still hot.

Key layouts: good, acceptable, bad

Recommended pattern: query dimensions first, entropy below them.

s3://bucket/{YYYY}/{MM}/{DD}/{HH}/{tranche}/{hash2}/{object-id}

- Time prefixes support ILM, support tickets, and incident scoping.
- tranche (for example 5-10 minute buckets) bounds the blast radius of a hot window.
- hash2 (two hex chars, 256-way) spreads a single tranche so peak obj/s does not pile into one leaf.

A tenant-aware variant is often better when multi-tenancy is real:

s3://bucket/{tenant}/{YYYY}/{MM}/{DD}/{HH}/{hash2}/{object-id}

Acceptable: pure hash fan-out when producers cannot emit useful time keys.

s3://bucket/{hash[0:2]}/{hash[2:4]}/{object-id}

Uniform and simple. Pair it with strict ILM and accept weaker time locality for ops and lifecycle.

Bad (call these out in design reviews):

```
s3://bucket/{sequential-id} # single leaf explosion
s3://bucket/{YYYY}/{MM}/{DD}/{id} # whole day in one or few leaves at LOSF day-scale
s3://bucket/{customerId}/{id} # hot tenants create hot leaves
s3://bucket/{uuid} # flat after the first char of entropy is ignored
```

Also bad for LIST IOPS: application code that lists without a tight prefix, recursive gateway listings, or SFTP-style recursive `ls` translated to unbounded `ListObjectsV2`. Prefixing only helps LIST when clients issue restrictive prefix requests. A whole-bucket list still walks the whole namespace; S3 prefixes limit returned keys, and delimiters only summarize deeper levels for the caller that asked for them.

What good partitioning changes in experiments

A warp-style load method that keeps paying off:

- LOSF PUT with intentionally flat keys (- -noprefix class worst case) and watch p95/p99 die as the leaf grows,
- LIST curves at 10k → 100k → 1M objects flat,
- same counts with partitioned keys: expect large factors of list improvement (often **5x-50x** class depending on depth and cache warmness),
- concurrent LIST + PUT on one flat leaf: expect material PUT degradation from directory metadata contention.

Partitioning is not cosmetics for pretty keys. It is how you keep XFS in the regime where `readdir` is cheap and background walks finish this week.

Threshold summary

One table to take into a design review. Two of these rows are values read from MinIO's own source, and the rest are operability judgements. The distinction matters, so the table names it.

Threshold	Value I hold to	Where it comes from	What exceeding it costs
Object subdirectories per leaf prefix	< 10 000 , alarm well before 50 000	My operability target, driven by XFS <code>readdir</code> and LIST cost	LIST latency degrades, <code>readdir</code> turns sequential, PUT contends on the same directory
Child prefixes per parent (community edition)	< 50 000 alert, forced stats compaction at 250 000	<code>scannerExcessFolders = 50000</code> and <code>dataScannerForceCompactAtFolders = 250_000</code> in <code>cmd/data-scanner.go</code>	Alert plus audit event. The scanner never aborts here, it just gets slower and the usage tree loses resolution
Child prefixes per parent (AIStor)	< 50 000 alert, errors logged at 500 000, prefix skipped at 5M	AIStor scanner documentation	Above 5M the scanner abandons the prefix for the whole cycle, so ILM and heal silently skip it

Threshold	Value I hold to	Where it comes from	What exceeding it costs
Objects per bucket	< ~100M	Operability ceiling, workload-dependent	Scan and ILM measured in weeks, listings expensive enough to need scheduling
Total cluster objects	No memory limit. Metadata is always disk-backed	MinIO keeps no global in-RAM metadata cache	Progressive I/O degradation rather than a cliff, which is why it is easy to miss
Useful prefix depth	2 to 4 levels	Trade-off, not a limit	Too flat builds monster directories. Too deep adds path cost and human pain for no capacity gain

Source-backed values and operability targets, kept apart on purpose.

The community-edition row is the one that misleads people most often. At 50 000 child prefixes MinIO emits an alert and an audit event, then **carries on scanning**. There is no abort path in the open-source scanner. So a cluster can sit far past the guardrail for months, look healthy on every dashboard, and still be paying for it in scan cycles that no longer complete inside the retention windows ILM is supposed to enforce. The absence of an error is not evidence that the layout is fine.

LIST IOPS cost model (why one hot leaf hurts)

LIST on one hot leaf hurts because the cost is per object and serial: a million-object leaf means a serial readdir over a large B+tree while the drives of one erasure set saturate and the rest of the cluster idles. For cold cache and inlined objects, a practical per-object cost on a primary drive is on the order of:

- parent readdir amortization: small,
- lookup/stat of the object directory: ~1 IOPS class,
- open/read `xl.meta`: ~1 IOPS class,
- **total ≈ 2-3 cold IOPS per listed object per primary drive** (I use ~2.5 as a rule of thumb).

Rough cluster-order model:

One detail decides the whole model. On a LIST walk MinIO does **not** read `xl.meta` from every drive of the set for every object. It picks one primary drive per object by name hash and only consults the quorum when it detects an inconsistency. So the `xl.meta` reads spread across the drives of the set. Call q the number of drives `xl.meta` is read from per listed object, $q = 1$ on the nominal walker path:

```
per-drive (cold)  ≈ 2.5 × q × objects_in_set / drives_in_set
cluster (cold)   ≈ 2.5 × q × N_objects
parent readdir    ≈ N_objects / 100    (XFS leaf format)
                  ≈ N_objects / 50     (B+tree, large directories)
```

Listing 1M objects: five layouts, same object count

Reference cluster for the arithmetic below: 60 NVMe drives in 4 erasure sets of 15, drives rated around 50k IOPS on 4 KiB random reads, all objects inlined, cold cache, walker path ($q = 1$). The full key hash picks the set, so even a flat prefix still spreads over all four sets.

Layout	Objects	Obj / EC set	Per-drive IOPS (cold)	Cluster IOPS	Wall clock	Why
(a) one leaf, 1k objects	1 000	250	~42	~2.5k	< 50 ms	Trivial, stays warm in cache.

Layout	Objects	Obj / EC set	Per-drive IOPS (cold)	Cluster IOPS	Wall clock	Why
(b) one leaf, 10k objects	10 000	2 500	~420	~25k	~100 ms	At the house target, still comfortable.
(c) 100 leaves x 10k	1 000 000	250k	~42k cumulative	~2.5M	~5 s at 10 concurrent LIST	Good partitioning. Each leaf fits page cache, listings parallelize.
(d) 1 000 leaves x 1k	1 000 000	250k	~42k cumulative	~2.5M	~3 s at 20 concurrent LIST	Best cache behaviour, cost moves to 1 000 S3 round trips.
(e) one leaf, 1M objects	1 000 000	250k per set	~42k	~2.5M	30 s to several minutes	Same IOPS on paper. Serial B+tree readdir over 250k entries, ~500 MB of xl.meta per set evicted from cache, 1 000 sequential list pages on one subtree.

Cold-cache LIST cost on a 60-drive cluster, 4 sets of 15, walker path. Orders of magnitude, not benchmarks.

Reading the table:

- **(c), (d) and (e) hold the same million objects and burn near-identical aggregate IOPS, yet behave nothing alike.** Many small leaves are parallelizable and cache-friendly. One monster leaf serializes `readdir`, serializes the multi-page S3 list loop, and pins the drives of one set while the rest of the cluster idles.
- **Case (e) is the "burned IOPS budget" incident.** A single badly placed LIST holds those drives under load for tens of seconds. PUTs landing on the same set see their latency climb and start timing out, MinIO answers 503 SlowDown, the client backs off for its configured window, and the platform looks frozen without a single error in the storage logs.
- **On (c) and (d) the bottleneck has already moved off the drives** and onto LIST issue rate: network latency times page count. That one you fix client-side with concurrent prefix-scoped listings.
- **The scanner walks the same tree.** In case (e), scanner plus one user LIST is immediate saturation. The excess-subfolder guardrails do not protect you here: they count *subfolders*, not the million *objects* sitting in one leaf.

If you hardened the list quorum, multiply

Regulated platforms often raise the listing quorum for consistency, which pushes `q` from 1 toward the full set width. On a set of 15 that is a factor of 15 on the `xl.meta` share of the cost, and `xl.meta` is what dominates on inlined small objects:

Layout	Per-drive IOPS, q = 15	Cluster IOPS	Wall clock, upper bound
(b) one leaf, 10k objects	~6.3k	~375k	~0.5 to 1 s
(c) 100 leaves x 10k	~630k	~37.5M	~1 to 2 min
(d) 1 000 leaves x 1k	~630k	~37.5M	~45 to 90 s
(e) one leaf, 1M objects	~630k	~37.5M	several minutes to tens of minutes

The arithmetic that matters: at 50k IOPS per drive, ~630k IOPS per drive is a **12 to 13 second floor of pure I/O saturation** before you count `readdir` and serial pagination. Even the well-partitioned layouts (c) and (d), a comfortable few seconds on the walker path, land in the minute range. The window where client PUTs and LIST or scanner traffic fight over the same drives widens by the same factor.

Treat that column as an upper bound and confirm it by measurement. Whether a hardened quorum really forces a read on every drive of the set for every listed object, or only a cheaper metadata comparison, depends on the validation path in your exact version. The real cost sits somewhere between the two tables. Settle it with `mc admin trace --call s3.ListObjectsV2` correlated against `iostat -xm` per drive of the set. If the full factor holds, consistency versus LIST cost becomes a design decision of its own, weighed against prefix depth and set width rather than assumed away.

Useful public checks: `mc admin trace` filtered on `ListObjects*`, `iostat -xm` during the list, controlled samples of directory entry counts on a data path (never blanket-`ls` a hot prod leaf without limits), and correlation of LIST storms with PUT errors and single-set disk util pegs.

What partitioning cannot save

Partitioning cannot save a workload whose daily small-object counts stay in the **10⁷-10⁸/day** band for long retention on fixed hardware: prettier keys stop being enough, and the fixes are data-model changes, not key changes. Batch business units into larger objects, revisit erasure set geometry, and bound upstream fan-out. At that shape the honest options are architectural:

- **Stop storing one S3 object per business unit.** Batch into archives, parquet, rollup blobs, or a compact journal+sink so MinIO sees larger objects. Keep a hot keyed store elsewhere if you need point reads.
- **Revisit set geometry.** Smaller erasure sets reduce per-object metadata amplification at the cost of higher storage overhead %. On LOSF, inode/IOPS often dominate raw % capacity, so higher overhead can still be the cheaper option. Pair this with the erasure-efficiency discussion in the companion post on small-file EC inversion.
- **Bound upstream fan-out.** Cloud-shaped "one request → millions of extracted objects" features assume elastic backends.
- **Do not expect compression of already-inlined scraps to fix LOSF.** CPU cost is per object; metadata IOPS count does not drop.
- **KMS/KES cost is roughly per object**, so crypto becomes a larger fraction of PUT latency on LOSF than on multi-MiB objects.

Monitoring checklist

The monitoring checklist for a LOSF MinIO cluster watches inode and byte capacity on every data and notification-spool mount, scanner progress and estimated full-scan time, mean object size versus used

bytes, and version and delete-marker growth. It also tracks list latency and the list IOPS to put IOPS ratio on heavy prefixes, PutObjectTagging against PutObject rate, disk utilization skew during lists, replication pending and failed gauges, and ILM backlogs.

- inodes *and* bytes on every MinIO data mount and on any notification spool mount (`df -h` and `df -i`),
- scanner progress / estimated full-scan time,
- object count growth vs used bytes (mean object size),
- versions per object and delete-marker growth when versioning is on,
- list latency for known heavy prefixes; ratio of list IOPS to put IOPS,
- PutObjectTagging rate against PutObject rate, since on inlined objects the two cost about the same,
- disk %util / await skew across nodes and EC sets during lists,
- replication pending/failed gauges on multi-site,
- ILM pending expiry / transition backlogs,
- filesystem geometry / maxpct awareness when inode headroom is tight.

Practical design contract I put in front of producers

The design contract I put in front of producers treats mean object size and peak obj/s as capacity inputs equal to TiB, holds keys to the leaf and fan-out targets under peak day growth, and scopes every LIST and lifecycle tool to a prefix. Versioning and site replication are multiplicative taxes, tags are set at PUT time, and when day-scale scans become unrealistic the data model changes before another rack is bought.

- Mean object size and peak obj/s are capacity inputs equal to TiB.
- Keys must satisfy the leaf and fan-out targets above under peak day growth for the full retention window.
- All LIST and lifecycle tools must be prefix-scoped. Bulk recursive list is a bug.
- Any API that expands one request into unbounded object creates needs a hard bound and an observable queue.
- Versioning and site replication are multiplicative taxes; clean noncurrent versions and delete markers on purpose.
- Tags are set at PUT time. Any post-ingest tag mutation counts against the write budget, and bulk retag campaigns need scheduling like a migration.
- If daily LOSF counts make day-scale scans unrealistic, change the data model before buying another rack of the same mistake.

Where this sits relative to the earlier posts

This post sits between the earlier MinIO articles on product fit, on the structural gaps versus databases, and on erasure coding as 15× replication, and it adds the filesystem substrate: XFS layout, inode capacity, scanner coupling and metadata write amplification on tag updates. "Sync mode will not save your RPO" then draws the multi-site consequence of the same scanner.

- [Stop using MinIO as a NoSQL database](#) - product-fit and the de-facto key-value antipattern.

- [MinIO and lots of small files](#) - structural gaps vs databases (no global index, sampling scanner, tiering interactions).
- [Erasure coding as 15x replication](#) - storage-efficiency inversion on inlined small objects.
- **This post** - filesystem substrate: XFS layout, inode capacity, scanner coupling, metadata write amplification on tag updates, and the prefix contract that keeps directories operable.
- [Sync mode will not save your RPO](#) - the multi-site consequence of the same scanner: failed replications are reconciled on a full sweep, so object count sets your DR RPO.

MinIO remains a strong fit for medium-to-large objects with S3 semantics on commodity hardware. LOSF is the envelope boundary. Prefix design is how you stay on the right side of that boundary for as long as the object model is still sane. Past that boundary, fix the model.

If free TiB still looks fine while PUT, LIST, scanner, or ILM already look like a capacity incident, the constraint is usually object count, directory shape, and erasure-set width rather than another shelf of drives. A [Data Platform Performance Audit](#) or a focused [Platform Sizing](#) review is the faster path than learning this on production traffic. [Book a 15-min intro call](#) to see which fits, or take one focused hour with an [Expert Call](#).

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 small files: when erasure coding becomes 15x replication](#)
- [MinIO and lots of small files: why it's not a database](#)
- [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.