Snapshot Visible, Restore Fails: How an S3 Lifecycle Rule Silently Breaks Your Elasticsearch Backups

Daniel Gleim · 2026-10-06



Snapshot Visible, Restore Fails: How an S3 Lifecycle Rule Silently Breaks Your Elasticsearch Backups

Introduction

Imagine you run a healthy Elasticsearch 9.2.1 cluster. Snapshot Lifecycle Management (SLM) takes a snapshot into your S3 bucket every night, and every morning the snapshot reports `SUCCESS` with all shards accounted for. You have never had a reason to doubt your backups.

Then you build a second cluster, point it at the same repository, and start a restore. It fails immediately with a `snapshot_missing_exception`, even though the very same snapshot shows up perfectly in the repository listing.

This is one of the more unsettling failure modes in Elasticsearch, because everything looks fine right up until the moment you actually need your data. The good news is that the cause is usually well understood, easy to identify, and completely preventable. Let's walk through what happens, why it happens, and how to keep it from happening to you.

Problem Description

You have an SLM policy writing nightly snapshots into an S3 repository. Listing the repository works exactly as expected:

© 2026 floragunn GmbH · search-guard.com

```
GET _snapshot/nightly-snapshots-003/_all
```

Asking for details about one specific snapshot also works, and the response looks reassuring:


```
{
  "snapshots" : [
    {
      "snapshot" : "snapshot-2026-06-11-00-00-00-utc-mivyqa0-sm2mw-ylrthncq",
      "uuid" : "_QbSq5jYQ8CxPfyLcPZ-pA",
      "version" : "9.2.0-9.2.1",
      "include_global_state" : true,
      "metadata" : { "policy" : "snapshot-003" },
      "state" : "SUCCESS",
      "start_time" : "2026-06-10T23:59:59.859Z",
      "end_time" : "2026-06-11T00:09:43.588Z",
      "failures" : [ ],
      "shards" : { "total" : 30, "failed" : 0, "successful" : 30 }
    }
  ],
  "total" : 1,
  "remaining" : 0
}
```


Thirty shards, zero failures, state `SUCCESS`.

One detail in that response is worth noting before you go further. SLM appends a random suffix to every snapshot it creates, so the real name is the full string `snapshot-2026-06-11-00-00-00-utc-mivyqa0-sm2mw-ylrthncq`, not just the date portion you configured in your policy. Copy the name exactly as the listing shows it:

```
POST /_snapshot/nightly-snapshots-003/snapshot-2026-06-11-00-00-00-utc-mivyqa0-sm2mw-
ylrthncq/_restore?pretty
```

And this is where it falls apart:

```
{
  "error" : {
    "root_cause" : [
      {
        "type" : "snapshot_missing_exception",
        "reason" : "[nightly-snapshots-003:snapshot-2026-06-11-00-00-00-utc-mivyqa0-sm2mw-
ylrthncq/_QbSq5jYQ8CxPfyLcPZ-pA] is missing"
      }
    ],
    "type" : "snapshot_missing_exception",
    "reason" : "[nightly-snapshots-003:snapshot-2026-06-11-00-00-00-utc-mivyqa0-sm2mw-
ylrthncq/_QbSq5jYQ8CxPfyLcPZ-pA] is missing",
    "caused_by" : {
      "type" : "no_such_file_exception",
      "reason" : "Blob object [backup/indices/CRlya0ZTSaew9OTZlxmyAw/meta-
iV_W4J0BJXbGAlFnZ1Wg.dat] not found: The specified key does not exist. (Service: S3, Status Code:
404)"
    }
  },
  "status" : 404
}
```


Two details stand out here. First, the snapshot is described as "missing" even though you can list it. Second, the underlying cause is a plain HTTP 404 from S3 for one specific object: a `meta-*.dat` file inside the `indices/` folder of the repository.

The `caused_by` block is your most useful diagnostic signal. Its presence tells you that Elasticsearch found the snapshot, resolved its UUID, began reading its metadata, and only then hit a missing object. A simple typo in the snapshot name produces a `snapshot_missing_exception` with nothing underneath it. When you see a blob level 404 nested inside, you are looking at a repository problem, not a request problem.

If you open the bucket in the AWS console and navigate to that path, the file really is gone. And when you retry with an older snapshot, you hit the same wall. This is not a single unlucky backup. Your repository has a systemic problem.

Root Cause Analysis

The short version: something other than Elasticsearch deleted objects from the bucket. In the case this article is based on, that something was an S3 lifecycle policy configured to expire objects older than 30 days.

To understand why the symptoms look so contradictory, it helps to know how Elasticsearch lays out a snapshot repository.

How a snapshot repository is organized

An Elasticsearch snapshot is not a single archive file. It is a tree of many small objects that reference each other:

Object	Purpose
`index-N`	The repository index. It lists every snapshot in the repository and which indices belong to each one. `N` is the generation number, which increases with every write.
`index.latest`	Holds the number of the current generation, so Elasticsearch knows which `index-N` file to read.
`snap-<snapshot-uuid>.dat`	The summary for one snapshot, held at the repository root. This is what the get snapshot API returns: state, timestamps, shard counts, and the list of indices.
`meta-<snapshot-uuid>.dat`	The global cluster metadata for one snapshot, also at the repository root.
`indices/<index-uuid>/meta-<snapshot-uuid>.dat`	Metadata for one index inside one snapshot, including mappings and settings.
`indices/<index-uuid>/<shard>/snap-<snapshot-uuid>.dat`	The list of segment files that make up one shard in one snapshot.
`indices/<index-uuid>/<shard>/__xxxx`	The actual Lucene segment data.

One detail is worth pausing on, because it trips people up when they browse the bucket: the `snap-` and `meta-` prefixes are reused at different levels of the tree with different meanings. A `snap-*.dat` at the root is a snapshot summary, while a `snap-*.dat` inside a shard folder is a list of segment files. Check the path, not just the file name.

This structure is what makes snapshots incremental. Two snapshots that share unchanged segments point at the same underlying data objects instead of storing two copies.

Why listing succeeds and restoring fails

Now the contradiction makes sense. When you call `GET _snapshot/<repo>/_all`, Elasticsearch reads `index.latest` to find the current generation, then reads the top level `index-N` file to see which snapshots exist. Asking about one specific snapshot adds a read of that snapshot's `snap-*.dat` summary at the repository root. Both of those files survived, so the snapshot is still listed, and the summary still says `SUCCESS`, because that status was recorded at the time the snapshot was taken.

A restore is a completely different operation. It has to walk down into the tree and read the per index metadata, then the per shard metadata, then the segment data. None of those files are touched by a listing. The moment the restore reaches the deleted `indices/<index-uuid>/meta-*.dat` object, S3 returns a 404, and Elasticsearch reports the snapshot as missing.

In other words, the listing is an inventory, not a health check. It tells you what the repository believes it contains, not what is actually still there.

© 2026 floragunn GmbH · search-guard.com

What the S3 lifecycle rule did

The setup in this case had retention configured twice, in two systems that knew nothing about each other:

Retention mechanism	Behavior
SLM policy with `expire_after: 30d`	Deletes snapshots that are older than 30 days, then removes only the data objects that no remaining snapshot still references.
S3 lifecycle rule expiring objects after 30 days	Deletes every object older than 30 days, with no knowledge of which objects are still in use.

That difference is the entire problem. SLM understands the reference graph. An S3 lifecycle rule only understands object age.

Because snapshots are incremental, a snapshot taken last night can easily depend on metadata and segment objects that were first written months ago and never changed since. Those objects are old by the calendar, but they are very much still in use. The lifecycle rule deleted them anyway, while the newer `index-N` file survived and kept pointing at them.

Elastic is explicit about this in the S3 repository documentation: you may use a lifecycle policy to adjust the storage class of objects, but you must not expire objects, and you must not move them to an unsupported storage class such as the Glacier classes. Doing either can cost you permanent access to your repository contents.

The same reasoning applies to anything else that writes to the bucket. Cleanup scripts, cost optimization tooling, a second cluster registered with write access, or bucket replication configured to copy objects into the repository can all cause the same kind of damage. Elasticsearch expects to be the only writer.

Solution

Recovery has two parts. First you find out how much of the repository is still intact. Then you remove the process that is damaging it so the problem stops recurring.

Set expectations honestly before you start: Elasticsearch cannot rebuild blobs that no longer exist. If the objects were permanently deleted, the affected snapshots are gone. What you can do is protect the snapshots that are still whole and make sure your future backups stay reliable.

Step 1: Stop writing to the repository

As soon as you suspect that a repository has been damaged, stop all snapshot activity against it. Continuing to write can rewrite the repository index and make the situation harder to reason about.

Pause Snapshot Lifecycle Management so that no policy fires while you investigate:

```
POST _slm/stop
```

This halts all SLM activity until you start it again with `POST _slm/start`. SLM has no endpoint for pausing an individual policy, and there is no schedule you can set that never fires, whether you configure a cron

expression or an interval. If you would rather keep your other schedules running, delete the policy that targets the damaged repository instead:

```
DELETE _slm/policy/snapshot-003
```

Deleting a policy only stops future runs. It does not touch snapshots that already exist, so you can recreate the policy once the repository is healthy again.

Then re-register the repository as read only on every cluster that has it attached:

```
PUT _snapshot/nightly-snapshots-003
{
  "type": "s3",
  "settings": {
    "bucket": "my-snapshot-bucket",
    "base_path": "backup",
    "readonly": true
  }
}
```

The `readonly` setting is what lets you safely inspect a repository from more than one cluster at a time.

Step 2: Check the integrity of the repository

Now find out exactly which snapshots are still usable. The verify repository integrity API scans the repository and reports anomalies that would prevent a restore:

```
POST _snapshot/nightly-snapshots-003/_verify_integrity
```

By default this check confirms that the objects a snapshot references are present. It does not read the contents of every data object. For a thorough check that also verifies the data itself, add the `verify_blob_contents` parameter. Be aware that this option downloads the entire repository, so plan for the time and the transfer cost:

```
POST _snapshot/nightly-snapshots-003/_verify_integrity?verify_blob_contents=true
```

A few practical notes on this API:

Consideration	What to keep in mind
Intended use	The API is designed for interactive troubleshooting by a human. Treat the response format as something that may change between versions, and avoid building automation on top of its exact output.
Cluster version	Run the check on a cluster where all nodes are on the same version. Results from a mixed version cluster are not reliable.
Large repositories	For very large repositories, consider attaching the repository read only to a separate single node cluster and running the check there, so the scan does not compete with production traffic.

© 2026 floragunn GmbH · search-guard.com

The output identifies the snapshots and indices whose objects are missing. Snapshots that come back clean are still restorable, so restore what you need from those right away.

You can also run the standard verification API, which checks that every node can reach the repository with the right credentials:

```
POST _snapshot/nightly-snapshots-003/_verify
```

This call confirms that your nodes can reach the repository with the right credentials and basic settings. It does not read every object or guarantee that all snapshots can be restored, so it can succeed even when snapshot data is missing. Use `_verify_integrity` when you need the deeper check.

Step 3: Remove the lifecycle rule from the bucket

This is the fix that actually solves the problem. Open the lifecycle configuration for the bucket in the AWS console, or inspect it from the command line:

```
aws s3api get-bucket-lifecycle-configuration --bucket my-snapshot-bucket
```

Remove any rule that expires or deletes objects in the path used by your repository. If the bucket is dedicated to Elasticsearch snapshots, remove the lifecycle configuration entirely:

```
aws s3api delete-bucket-lifecycle --bucket my-snapshot-bucket
```

Transition rules deserve the same scrutiny, though the picture there is more nuanced than a simple ban. Some storage classes are fine for a snapshot repository and some will break restores:

Storage class	Suitability
`standard`, `standard_ia`, `onezone_ia`	Supported. Objects stay available for immediate reads.
`reduced_redundancy`	Elasticsearch accepts this value, but AWS recommends against using it. It offers lower durability than `standard` and no longer costs less, so choose `standard` instead.
`glacier_ir` (Glacier Instant Retrieval)	Supported, because it still serves reads immediately.
`intelligent_tiering`	Supported, but only if you leave the optional Archive Access and Deep Archive Access tiers switched off. Turning those on can cost you access to the repository.
Glacier Flexible Retrieval, Glacier Deep Archive	Not supported. These tiers need a restore request before an object can be read, which breaks snapshot restores.

The rule of thumb is simple: a repository object has to be readable the instant Elasticsearch asks for it. Any tier that requires a thaw step first is off the table.

© 2026 floragunn GmbH · search-guard.com

Step 4: Let SLM own retention

With the bucket rules gone, hand retention back to the system that understands the reference graph. A policy with a retention block covers both age and count:

```
PUT _slm/policy/snapshot-003
{
  "schedule": "0 0 1 * * ?",
  "name": "<snapshot-{now/d{yyyy-MM-dd-HH-mm-ss}}-utc>",
  "repository": "nightly-snapshots-003",
  "config": {
    "indices": ["*"],
    "include_global_state": true
  },
  "retention": {
    "expire_after": "30d",
    "min_count": 5,
    "max_count": 50
  }
}
```

The `name` field uses date math, so each run produces a name based on the current date, and SLM adds its random suffix on top. That is where a name like `snapshot-2026-06-11-00-00-00-utc-mivyqa0-sm2mw-ylrthncq` comes from.

Here is what each retention setting does:

Setting	Explanation
`expire_after`	How long a snapshot is kept before it becomes eligible for deletion.
`min_count`	The number of snapshots always kept, even after they pass `expire_after`. This protects you when snapshots stop being created for a while.
`max_count`	The upper limit on retained snapshots. The oldest ones are removed first once the limit is reached.

The `retention` block decides what gets deleted, but a separate cluster level task decides when the deletion actually happens. That task is controlled by two cluster settings:

Setting	Explanation
`slm.retention_schedule`	When the retention task runs. It defaults to `0 30 1 * * ?`, which is daily at 01:30 UTC.
`slm.retention_duration`	How long the task is allowed to spend deleting snapshots in one run. It defaults to `1h`.

Both settings have sensible defaults, so retention works out of the box. It is still worth knowing that they exist, because a policy whose snapshots never seem to expire is usually a sign that the retention task is not running, rather than a problem with the policy itself. You can check on it with `GET _slm/stats` and trigger it by hand with `POST _slm/_execute_retention`.

When SLM deletes a snapshot, it removes only the data objects that no remaining snapshot references. Shared segments stay exactly where they are, which is why storage growth stays reasonable without any help from a lifecycle rule.

© 2026 floragunn GmbH · search-guard.com

Step 5: Give additional clusters read only access

Registering a second cluster against an existing repository is a normal and supported thing to do, as long as only one cluster is allowed to write. Register the repository on the second cluster with `readonly` set to `true`:

```
PUT _snapshot/nightly-snapshots-003
{
  "type": "s3",
  "settings": {
    "bucket": "my-snapshot-bucket",
    "base_path": "backup",
    "readonly": true
  }
}
```

You can confirm that clusters are not accidentally sharing a writable repository by comparing repository UUIDs:

```
GET _snapshot/_all
```

Each repository carries a UUID that is stored in the repository itself and recorded in the cluster state. Duplicate UUIDs in that list are a documented signal that two registrations point at the same storage, so treat them as a prompt to review your repository registrations.

Elasticsearch 9.3.0 hardened one related area, and it is worth being precise about what that change does and does not cover. From that version onward, the S3 repository backs its linearizable register with S3 conditional writes, using the `If-None-Match` and `If-Match` headers. That mechanism protects the repository when two writers compete for it, and it stops rather than compounding the damage.

It would not have helped in this case. Conditional writes guard against a second cluster writing to the repository at the same time as yours. A lifecycle rule deletes objects outside Elasticsearch entirely, so there is no conflicting write for the register to catch, and no version of Elasticsearch can bring back objects that are already gone. The cluster behind this article runs 9.2.x, but upgrading would not have prevented this particular failure. Removing the external writer remains the fix.

Step 6: Start fresh if the damage is extensive

If the integrity check shows that most of your snapshots are affected, the cleanest path forward is a new repository. Create a new bucket or a new `base_path`, register it, and take a fresh full snapshot:

```
PUT _snapshot/nightly-snapshots-004
{
  "type": "s3",
  "settings": {
    "bucket": "my-snapshot-bucket-v2",
    "base_path": "backup"
  }
}
```

© 2026 floragunn GmbH · search-guard.com

```
PUT _snapshot/nightly-snapshots-004/initial-full-snapshot?wait_for_completion=true
{
  "indices": "*",
  "include_global_state": true
}
```

Keep the old repository read only for as long as you still need the snapshots that survived, then retire it.

Step 7: Validate a new bucket before you trust it

If you are setting up storage that is not AWS S3 itself, for example a S3 compatible object store, the repository analysis API is worth running before the repository goes into service. It writes and reads test data to confirm that the store behaves correctly under the access patterns Elasticsearch relies on:

```
POST _snapshot/nightly-snapshots-004/_analyze?blob_count=10&max_blob_size=1mb
```

Start with modest parameters, since the analysis can move a substantial amount of data. As with the integrity API, treat this as an interactive tool rather than something to automate.

Takeaways

A green snapshot status is not proof of a working backup. The `SUCCESS` state records what happened when the snapshot was taken. It says nothing about whether the objects it depends on are still present today. Only an integrity check or an actual restore can tell you that.

Elasticsearch must be the only writer to the repository. Do not run any process that deletes or overwrites repository objects or moves them into unsupported archival tiers. That rules out cleanup scripts, cost optimization tooling, and a second cluster with write access. If you use S3 lifecycle rules at all, restrict them to supported storage class transitions and confirm what they do to your snapshot objects before you enable them.

Configure retention in exactly one place. SLM knows which objects are still referenced by which snapshots. Storage level rules only know how old an object is. When both are active, the storage rule eventually deletes something that a recent snapshot still needs.

Storage class transitions carry the same risk as deletion. Moving snapshot objects into an archival tier makes them unavailable for the immediate reads that a restore performs. Keep your repository objects in a storage class that supports direct retrieval.

Read the whole error, not just the first line. A `snapshot_missing_exception` on its own usually means the name you asked for does not exist, which is often just a missing SLM suffix. The same exception with a blob level 404 nested in `caused_by` means Elasticsearch found the snapshot and could not read its contents. Those are very different problems.

Test restores on a schedule. A backup you have never restored is a hypothesis. Restore into a staging cluster regularly, and run `_verify_integrity` against your repository from time to time. Both give you the chance to find damage while you still have options.

© 2026 floragunn GmbH · search-guard.com

Enable versioning on the bucket. With versioning turned on, a deletion creates a delete marker instead of destroying the object, and the previous version stays recoverable. This applies to lifecycle expiration too, which is a useful safety net. One caveat completes the picture: a noncurrent version expiration rule will permanently remove those older versions, so check that you have not configured one on a bucket you are counting on for recovery. Adding S3 Object Lock on top gives you a hard guarantee that objects cannot be deleted or overwritten during a retention period.

Back up the repository itself. Elasticsearch cannot repair a damaged repository. The only reliable route back to a fully working state is a copy of the repository taken before the damage occurred, created with the documented repository backup process rather than live replication.

Reliable backups are only one part of keeping an Elasticsearch cluster trustworthy, and knowing who touched your data matters just as much as knowing you can restore it. **Explore how Search Guard adds authentication, fine grained access control, and audit logging to your Elastic Stack** so that your cluster is protected in production, not just recoverable after the fact.

This article was published on the Search Guard blog: https://search-guard.com/blog/snapshot-visible-restore-fails-how-an-s3-lifecycle-rule-silently-breaks-your-elasticsearch-backups/
Search Guard is Security and Alerting for Elasticsearch and Kibana — search-guard.com · Start a free trial