Spark Output Committers on S3, MinIO and Ceph: Stop Committing by Rename

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

https://pepitedata.com/spark-s3-output-committers/

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



A Spark job using Hadoop's classic FileOutputCommitter on Amazon S3, MinIO or Ceph RGW can finish its last task and then spend minutes publishing the output. That gap is the output commit. FileOutputCommitter was designed for filesystems with cheap, atomic rename. For ordinary S3 buckets and compatible endpoints, the S3A connector implements rename by copying objects to new keys and deleting the originals. Managed distributions may already select a different committer.

The one-line version: **for plain file writes through S3A, replace rename-based commits with an S3A committer and verify the runtime selection. For atomic table updates, use a supported Apache Iceberg or Delta Lake commit path. Removing rename does not make a directory of files a transaction.**

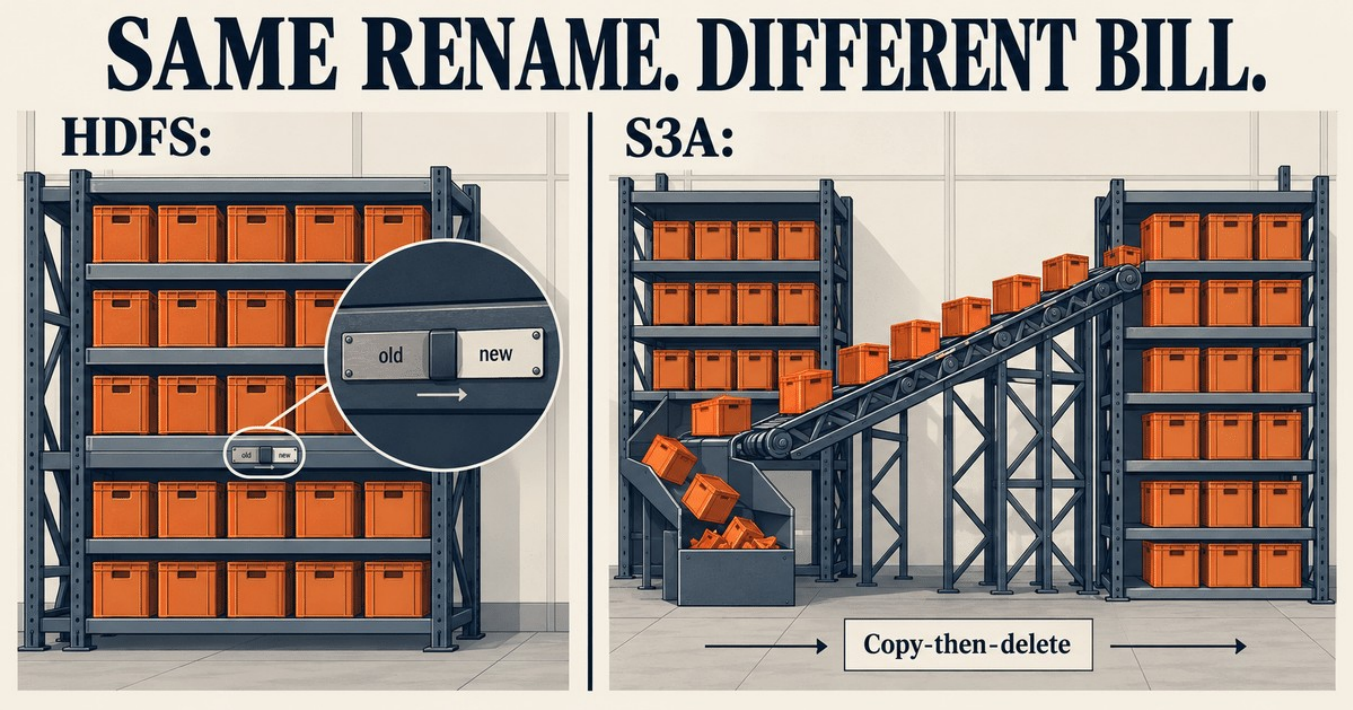
Why does Spark stall after its last task?

A Spark job on S3 stalls after its last task because the classic Hadoop FileOutputCommitter publishes output by renaming directories, and the S3 API has no rename. The Hadoop S3A connector emulates rename by copying each object and deleting the original, so commit time grows with the volume of data written. Spark's cloud integration guide rates S3A directory rename as O(data) and unsafe.

With commit algorithm version 1, the rename happens twice. Each task commit moves its attempt directory into a job directory, then the job commit moves everything again into the destination, from the

driver. On HDFS both steps are metadata operations. On S3, MinIO or Ceph RGW, each one is a server-side copy of every object followed by a delete, plus the listings needed to find them.

This is why the Spark UI can look finished while the job is not. Every write task reports complete, the executors sit idle, and the driver is still copying. In the Spark event log, the commit shows up as the gap between the end of the write stage and the end of the SQL execution that contains it.



On HDFS a rename changes metadata. On S3, MinIO or Ceph RGW the same rename copies every object to the new path, then deletes the original.

Rename-based commit exposes partial output

Committing by rename on S3 is unsafe because the emulated rename is not atomic. A copy-then-delete sequence that fails partway leaves some objects at the destination and others still under the temporary path, and a reader can see a half-published dataset. The Hadoop S3A committer documentation states that committing work by renaming directories on S3 is "unsafe as well as horribly slow".

Commit algorithm version 2 trades that risk for a worse one. It renames task output straight into the destination at task commit, so if a task commit fails partway and another attempt commits, output from the first attempt can end up mixed into the result (MAPREDUCE-7282). Spark made version 1 its default again in Spark 3.0.2 and 3.1.0 (SPARK-33019). A job that still sets `mapreduce.fileoutputcommitter.algorithm.version=2` to speed up S3 writes buys a faster commit and an output it cannot trust.

The S3A committers, compared

The Hadoop S3A connector ships three rename-free committers: directory, partitioned and magic. All three upload data as multipart uploads and postpone completion until job commit. Files become visible individually as their uploads complete; there is no atomic publication of the whole dataset. The committers differ in staging requirements and conflict handling, and first shipped in Hadoop 3.1.0.

Committer (`fs.s3a.committer.name`)	Where task output waits	What it requires	Best fit
`file` (classic, the default)	Temporary directories, then renamed	Nothing	HDFS only; unsafe and O(data) on S3
`directory` (staging)	Executor local disk, uploaded at task commit	A shared cluster filesystem such as HDFS for the pending-upload lists	Clusters that already run HDFS
`partitioned` (staging, Spark only)	Executor local disk, uploaded at task commit	Same as directory	Partitioned tables where only the partitions written should be replaced
`magic`	In the bucket, as multipart uploads not yet completed	A consistent S3 store; enabled by default since Hadoop 3.3.1	Spark on Kubernetes and any cluster without HDFS

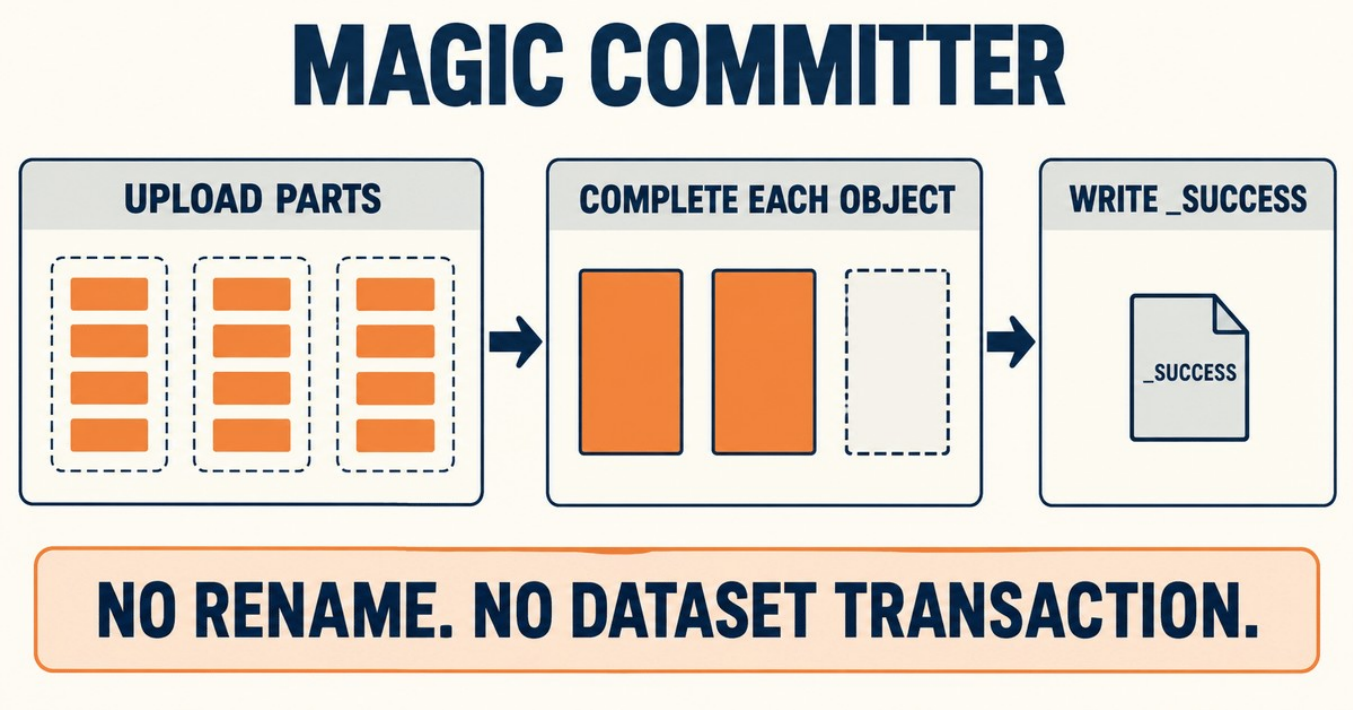
Sources: Hadoop S3A committers documentation; Spark "Integration with Cloud Infrastructures".

The staging committers come from production code at Netflix. Their weak point on Kubernetes is the shared filesystem: the pending-upload lists must sit somewhere every task and the driver can read, and a local directory breaks the job commit. That requirement is what makes the magic committer the natural choice when the cluster has no HDFS.

How the magic committer works

The magic committer uses deferred multipart-upload completion to avoid copying output during commit. Tasks upload parts for final destination keys and persist the metadata needed to complete them. At job commit, the driver completes those uploads individually. Readers can see a subset while commit is in progress; use a table format when readers need one atomic snapshot.

1. A task opens `dest/__magic/.../part-00000.parquet`. S3A starts a multipart upload for `dest/part-00000.parquet` and streams the parts while the task runs.
2. When the task closes the file, S3A saves the upload ID and part tags in a small `.pending` file instead of completing the upload.
3. Task commit gathers the task's pending files into one `.pendingset` in the job attempt directory. A failed or aborted task has its uploads aborted instead.
4. Job commit reads every pendingset, sends one completion request per file, writes a JSON _SUCCESS file and deletes the `__magic` directory.



Tasks upload parts without publishing the final objects. Job commit completes each object separately, then writes the success marker. This avoids rename but does not publish a dataset atomically.

Job commit now costs one small request per file, whatever the file size. The Hadoop documentation still calls the magic committer the least mature of the three, and adds that now S3 is consistent, there are fewer reasons not to use it. On Kubernetes it is the committer I would start from.

No dataset transaction. A successful multipart completion publishes one object atomically. If a later completion fails, some objects may already be visible while rollback is attempted; the cleanup can also fail. Use a fresh output prefix per run and release it to downstream readers only after verified job success. A reusable _SUCCESS marker alone does not isolate readers or concurrent writers.

Configuring Spark to use the magic committer

For upstream Spark file writes, an S3A committer requires compatible Spark and Hadoop dependencies plus the settings below. The integration exists in builds using Hadoop 3.1 or later; use a supported release for production. Select the committer explicitly and verify it at runtime, because managed distributions and table writers can use different commit paths. The Parquet binding applies to Parquet output.

```
spark.hadoop.fs.s3a.committer.name       magic
spark.sql.sources.commitProtocolClass
org.apache.spark.internal.io.cloud.PathOutputCommitProtocol
spark.sql.parquet.output.committer.class
org.apache.spark.internal.io.cloud.BindingParquetOutputCommitter
```

On Hadoop releases older than 3.3.1, also set `spark.hadoop.fs.s3a.committer.magic.enabled=true`. The same settings apply to MinIO and Ceph RGW through the S3A endpoint configuration; the committer does not care which server answers the S3 API, only that it is consistent.

Use the `spark-hadoop-cloud` module matching the Spark version and Scala binary version in the image. Keep `hadoop-aws` and the AWS SDK aligned with the bundled Hadoop release. Verify that `mapreduce.outputcommitter.factory.scheme.s3a` resolves to `org.apache.hadoop.fs.s3a.commit.S3ACommitterFactory`, normally supplied by Hadoop defaults. Magic-path recognition being enabled since Hadoop 3.3.1 does not mean the magic committer is selected by default.

Dynamic partition overwrite is not supported

The upstream Hadoop S3A committers described here do not support `spark.sql.sources.partitionOverwriteMode=dynamic`: Spark fails the job with "PathOutputCommitter does not support dynamicPartitionOverwrite". The partitioned staging committer with `fs.s3a.committer.staging.conflict-mode=replace` covers the usual need, replacing only the partitions a job writes, but it brings the shared-filesystem requirement back. A supported table format provides its own overwrite semantics; check its catalog and concurrency requirements.

Hadoop's newer manifest committer (Hadoop 3.3.5) does support dynamic overwrite, with Spark 3.5.0 or later, but it targets Azure ADLS Gen2 and Google Cloud Storage and still relies on rename. It is not an option for S3, MinIO or Ceph.

How to prove which committer actually ran

For upstream Hadoop file writers, _SUCCESS is a useful committer diagnostic: FileOutputCommitter writes an empty marker; S3A committers write JSON containing the `committer` name. Read the marker from a fresh output prefix and compare it with driver logs and effective configuration. JSON alone does not identify the implementation, and this test does not cover every vendor committer or table writer.

Three more signals confirm it from the outside:

- **Commit time in the event log.** Measure the gap between the end of the write stage and the end of its SQL execution. Rename-free commit still grows with file count, metadata processing, request latency and throttling; there is no universal seconds-only bound.
- **Request mix on the storage side.** A rename-based commit produces a burst of server-side COPY and DELETE requests at the end of the job, visible in MinIO or RGW access logs and request metrics. A committer-based job produces completion requests instead.
- **The commit algorithm setting.** Search job configurations and cluster defaults for `algorithm.version=2`; it tends to survive in old templates long after its author has left.

For plain file pipelines under your control, enable `mapreduce.fileoutputcommitter.marksuccessfuljobs` and validate the JSON `committer` field against an allowlist. Treat a missing marker as an investigation, since marker creation may be disabled. An existing marker can be stale if a prefix is reused. Iceberg, Delta Lake and managed committers need checks appropriate to their own commit path.

Clean up incomplete uploads without aborting active jobs

Failed S3A jobs can leave incomplete multipart uploads that consume storage without appearing as completed objects. AWS S3 and Ceph RGW document lifecycle cleanup with `AbortIncompleteMultipartUpload`. Check the exact MinIO edition and release before relying on that lifecycle action; MinIO also has server-wide stale-upload cleanup, with a documented default expiry of 24 hours and scan interval of 6 hours.

Hadoop also provides a manual sweep scoped to a path, `hadoop s3guard uploads -list` and `hadoop s3guard uploads -abort -days 2 s3a://bucket/path`, useful after an incident. Keep the age threshold, in the CLI and in any lifecycle rule, above your longest job runtime, or the cleanup aborts uploads that a running job still needs.

Prefer separate output prefixes for concurrent jobs. Hadoop documents `fs.s3a.committer.abort.pending.uploads=true` as the default: a finishing job can abort another job's pending uploads under the same destination. Setting it to `false` avoids that sweep but leaves cleanup to another process and does not make conflicting writes safe. Coordinate writers and use unique filenames.

MinIO and Ceph RGW: what to check first

The magic committer needs a store with consistent reads and listings, which AWS S3 has provided for all buckets since December 2020. MinIO documents strict read-after-write and list-after-write consistency, provided its drives use XFS, ZFS or Btrfs rather than ext4 or NFS. Ceph documents eventual consistency for multisite replication between zones, but I found no RGW document stating single-site read-after-write in S3 terms, so test it on your release.

In a multisite deployment, point Spark at the zone that owns the bucket rather than at a replica, so the job commit and the readers that follow see the same state. And count the files: a job that writes 200,000 small files issues 200,000 completion requests at job commit, which moves the small-file problem from the write phase to the commit phase.

On EMR, the release and connector decide

On Amazon EMR, choose by release and filesystem connector rather than applying one universal setting. EMR 7.10 and later use S3A as the default S3 connector. AWS documents a separate EMR `MagicCommitProtocol` using `MagicV2Committer` for S3A, enabled by default. This vendor implementation has different behavior from the upstream magic committer configured above.

For EMRFS, the S3-optimized committer arrived in EMR 5.19.0 and became the default in 5.20.0; support expanded beyond Parquet in 6.4.0. EMRFS dynamic partition overwrite uses its separate S3-optimized commit protocol. EMR's S3A MagicCommitProtocol also handles dynamic overwrite: AWS describes static inserts publishing at task commit and dynamic inserts publishing during job commit. Verify the connector, actual protocol class and driver logs instead of assuming that upstream S3A restrictions or its marker format apply unchanged.

Iceberg and Delta Lake commit through metadata

Table formats remove the rename problem at the source. Apache Iceberg writes immutable data files to unique paths and commits by atomically updating the catalog's reference to table metadata, with optimistic concurrency, so no listing or rename is involved. Delta Lake commits by adding a new entry to its `_delta_log`. For table writes, the S3A committer question no longer applies.

Each format has one condition on object storage. Iceberg's atomic metadata-reference update must come from a compatible catalog (REST, JDBC, Hive Metastore, Nessie or AWS Glue): the Iceberg specification calls the file-system commit scheme based on rename unsafe in object stores. Delta Lake's documentation states that concurrent writes to S3 must come from a single Spark driver by default, because S3 does not provide mutual exclusion; multi-cluster writes need the `S3DynamoDBLogStore`. For the documented default S3 LogStore on MinIO or Ceph, plan for one writing driver per Delta table unless you have validated a supported coordination or catalog-managed commit implementation. A catalog name alone does not establish safe concurrency.

Committers do not fix small files

A committer changes how files are published, not how many files a job writes. A job that writes one file per task per partition value still produces thousands of small objects, and with the magic committer each one costs a multipart upload and a completion request. The file count is decided earlier, by the partitioning of the last stage before the write.

- **Shape the last stage.** A `REBALANCE` hint, or `REPARTITION` by the table's partition columns, before the write can reduce file counts, but does not guarantee a target file size. With Adaptive Query Execution on (the default since Spark 3.2.0), the advisory size is `spark.sql.adaptive.advisoryPartitionSizeInBytes`, 64 MB by default. That is a shuffle-partition

target, not a Parquet file-size guarantee; the default `parallelismFirst=true` can make coalescing favor parallelism over that target.

- **Cap the large ones.** `spark.sql.files.maxRecordsPerFile` limits rows per file (0, no limit, by default). It prevents oversized files; it does nothing for small ones.
- **Compact what accumulates.** Tables that receive many small appends need scheduled compaction: `CALL catalog.system.rewrite_data_files('db.table')` on Iceberg, `OPTIMIZE table` on Delta Lake. Run it as its own job, with its own resource budget.

What should a Spark write-path review check?

When I review a Spark platform that writes to S3, MinIO or Ceph RGW, the write path gets five checks: which committer the _SUCCESS files name, how long job commit takes against output size, whether commit algorithm version 2 is set anywhere, how many files each job writes per partition, and whether failed jobs leave multipart uploads behind. Most fixes are configuration plus one guard step.

The write path is one part of the measurement discipline in the performance and cost engineering guide: find where the time goes before tuning anything. If your Spark jobs on object storage are slow at the end, unreliable after failures or heavy on requests, a Flash Architecture Review covers the write path in depth, and a data platform audit covers it together with shuffle, Kubernetes scheduling and storage. You can also book a 15-minute introductory call.

FAQ

Which Spark output committer should I use with MinIO?

Use the S3A magic committer for plain file writes to MinIO: set `fs.s3a.committer.name=magic`, bind Spark with PathOutputCommitProtocol and BindingParquetOutputCommitter, and add the spark-hadoop-cloud module. MinIO's documented read-after-write consistency, on XFS, ZFS or Btrfs drives, meets the committer's requirement. For tables, write through Apache Iceberg or Delta Lake instead.

Does the magic committer publish an atomic dataset?

No. The S3A magic committer completes each file's multipart upload separately at job commit, so readers listing the destination during commit can see only part of the output. Per-object atomicity is not dataset atomicity. Use an isolated run prefix with an explicit reader handoff, or Apache Iceberg or Delta Lake with a validated catalog and storage integration.

Is the FileOutputCommitter algorithm version 2 safe on S3?

No. Commit algorithm version 2 renames task output directly into the destination, so a task commit that fails partway can leave output that mixes with a retried attempt (MAPREDUCE-7282). On S3 the rename is also a non-atomic copy and delete. Spark returned to version 1 by default in 3.0.2 and 3.1.0; the fix on S3 is an S3A committer.

Does the magic committer support dynamic partition overwrite?

No, not in upstream Hadoop: the S3A committers, magic included, do not support Spark's `partitionOverwriteMode=dynamic`. The partitioned staging committer provides partition-scoped replacement through conflict mode `replace` and needs a shared cluster filesystem. Amazon EMR's MagicCommitProtocol, a separate vendor implementation, supports dynamic overwrite. For table writes, use the table format's documented overwrite operation.

Do I need an S3A committer with Iceberg or Delta Lake?

Not for table writes through their supported integrations. Apache Iceberg publishes a new metadata reference through a compatible catalog, and Delta Lake publishes a transaction-log entry. Readers must use the table format to see an atomic snapshot. For the documented default S3 LogStore, Delta concurrent writes must come from one Spark driver unless a supported multi-writer implementation is configured.

How do I know which committer wrote my Spark output?

Read the _SUCCESS marker from a fresh output prefix. For upstream Hadoop plain file writers, an empty file points to the rename-based FileOutputCommitter, and the S3A committers write JSON whose `committer` field names the committer. Confirm the result in driver logs and effective configuration. Other committers can also write JSON, marker creation can be disabled, and table-format writes require their own commit checks.

Sources

- Apache Hadoop, Committing work to S3 with the S3A Committers and S3A Committers: Architecture and Implementation
- Apache Spark, Integration with Cloud Infrastructures and Performance Tuning
- Apache JIRA: MAPREDUCE-7282 (v2 commit algorithm), SPARK-33019 (Spark default back to v1), HADOOP-17483 (magic committer enabled by default, Hadoop 3.3.1), SPARK-40034 (dynamic overwrite with the manifest committer)
- Apache Hadoop, The Manifest Committer for Azure and Google Cloud Storage and s3guard uploads command
- AWS, Amazon S3 now delivers strong read-after-write consistency, December 2020; Configuring a lifecycle rule to delete incomplete multipart uploads
- AWS, EMRFS S3-optimized committer, its requirements, EMR MagicCommitProtocol, S3A by default from EMR 7.10 and the EMRFS S3-optimized commit protocol
- Apache Iceberg, Reliability, Table specification and Spark procedures
- Delta Lake, Storage configuration and Optimizations
- MinIO, Distributed MinIO consistency guarantees and S3 API compatibility and AIStor stale multipart-upload settings; Ceph, RGW S3 API support and RGW metrics

Reviewed against Apache and vendor documentation on 27 October 2026. The upstream configuration follows Hadoop 3.4.1 documentation; managed-distribution defaults must be checked on the installed release. This article explains documented behavior, not a benchmark performed for this review. I have no affiliation with any project or vendor cited here, and none of them reviewed or endorsed this article.

Related posts

- Performance and cost engineering for data platforms
- Spark on Kubernetes reserves CPU it never uses
- Apache Spark smoke tests on Kubernetes
- Beyond Parquet and Avro: choosing a data file format by access pattern

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