MongoDB Backups

What the Documentation Gets Wrong

FEBRUARY 25, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

Most MongoDB backup guides end at `mongodump`. The real complexity starts at `mongorestore`.

I ran self-hosted MongoDB replica sets in production for over a decade, first on six EC2 m5d.xlarge instances serving 34 e-commerce websites across the US and EU, now on a lean Docker Swarm stack across two continents for $166/year. Over 3,650 daily backups. Zero data loss. Two corrupted dumps caught by restore testing that would have been catastrophic if discovered during an actual failure.

This is the backup and restore guide that would have saved me a lot of sleepless nights.

THE BACKUP PIPELINE THAT SURVIVED A DECADE

The principle is simple. The execution is where people get hurt.

3 copies of your data: Primary + Secondary + Off-site backup. **2 different media:** Live replica set + compressed archive. **1 off-site:** Shipped to a different provider, different region.

Here's the actual pipeline:

1. Always dump from the secondary. Never the primary. A mongodump against a busy primary will degrade write performance. Your secondary exists for exactly this purpose.

2. Always capture the oplog. This is the detail most guides skip. Without it, your backup is a snapshot of whatever moment the dump started. With it, you can replay operations forward to any specific second.

Someone runs a bad migration that corrupts your products table at 2:47 PM? Without oplog capture, you're restoring to whenever your last dump completed, maybe 3 AM. With it, you restore to 2:46 PM. That's the difference between losing a day of data and losing a minute.

3. Use `--gzip` built into mongodump. This is worth emphasizing. MongoDB's built-in gzip compresses the data as it streams directly from the database into the archive, no intermediate uncompressed file, no extra disk space needed. My production database was 12GB uncompressed. The gzip archive: **1.5GB.** That's an 87.5%

reduction, streamed directly to S3 without ever touching 12GB of disk. For daily backups shipping off-site, this is the difference between a backup that finishes in minutes and one that saturates your network for an hour.

4. Ship off-site immediately. Compressed and encrypted. A backup sitting on the same server as your database isn't a backup, it's a second copy of the same single point of failure.

5. Retain strategically. 7 daily + 4 weekly + 12 monthly. Storage is cheap. The dump from 3 months ago that you deleted might be the only clean copy before a slow data corruption you didn't notice.

6. Test your restores. Monthly. Non-negotiable. Over ten years I caught two corrupted dumps, two out of roughly 3,650. That's a 99.95% success rate. The 0.05% would have been invisible without restore testing, and catastrophic if I'd discovered it during an actual failure.

A backup you've never restored is a hope, not a strategy.

THE BACKUP SCRIPT

Here's a simplified version of the script I've been running in production. The key design decision: it saves a **collection inventory file** alongside every backup. I'll explain why this matters in a moment, it solves a problem that has cost me and many others serious pain.

```
#!/bin/bash
set -e

# --- Configuration ---
MONGO_HOST="mongodb-secondary.internal:27017"   # Always dump from secondary
MONGO_USER="backup_user"
MONGO_PASS="your_password"
MONGO_AUTH_DB="admin"
MONGO_DB="products"
S3_BUCKET="s3://your-bucket/mongo_backups"

# --- Timestamp ---
TIMESTAMP=$(date +"%Y%m%d_%H%M%S")
S3_BACKUP="${S3_BUCKET}/${MONGO_DB}/${TIMESTAMP}.dump.gz"
S3_LATEST="${S3_BUCKET}/${MONGO_DB}/latest.dump.gz"
S3_COLLECTIONS="${S3_BUCKET}/${MONGO_DB}/${TIMESTAMP}.collections.txt"
S3_COLLECTIONS_LATEST="${S3_BUCKET}/${MONGO_DB}/latest.collections.txt"

echo "[$(date)] Starting backup of ${MONGO_DB}..."

# --- Step 1: Save collection inventory ---
# This file saves you at 2 AM. It lists every collection
# in the database at backup time, because you CANNOT inspect
# the contents of a gzip archive after the fact.
mongosh --quiet \
  --host "$MONGO_HOST" \
  --username "$MONGO_USER" \
  --password "$MONGO_PASS" \
  --authenticationDatabase "$MONGO_AUTH_DB" \
  --eval "db.getSiblingDB('${MONGO_DB}').getCollectionNames().forEach(c => prin
  > /tmp/collections_${TIMESTAMP}.txt

COLLECTION_COUNT=$(wc -l < /tmp/collections_${TIMESTAMP}.txt)
echo "[$(date)] Found ${COLLECTION_COUNT} collections"

aws s3 cp /tmp/collections_${TIMESTAMP}.txt "$S3_COLLECTIONS" --quiet
aws s3 cp /tmp/collections_${TIMESTAMP}.txt "$S3_COLLECTIONS_LATEST" --quiet

# --- Step 2: Stream backup directly to S3 ---
# No intermediate file. 12GB database → 1.5GB gzip → straight to S3.
mongodump \
  --host "$MONGO_HOST" \
  --username "$MONGO_USER" \
  --password "$MONGO_PASS" \
  --authenticationDatabase "$MONGO_AUTH_DB" \
  --db "$MONGO_DB" \
```

```
  --oplog \
  --gzip \
  --archive \
  | aws s3 cp - "$S3_BACKUP"

# --- Step 3: Copy as latest ---
aws s3 cp "$S3_BACKUP" "$S3_LATEST" --quiet

rm -f /tmp/collections_${TIMESTAMP}.txt
echo "[$(date)] Backup complete: ${S3_BACKUP} (${COLLECTION_COUNT} collections)
```

Schedule it with cron, and every night you get a timestamped backup plus a `latest` alias, both with a matching collection inventory. The `latest.dump.gz` / `latest.collections.txt` convention means your restore scripts always know where to look.

My original production version of this script ran for years on a replica set across three `m5d.xlarge` instances, piping directly to S3. The entire backup, 12GB of database compressed to 1.5GB, completed in minutes without ever writing a temporary file to disk.

THE `--NSINCLUDE` BUG NOBODY TALKS ABOUT

This one cost me hours. And it turns out I'm not the only one.

In production, you almost never restore an entire database. You restore specific collections. Maybe someone ran a bad script on the products table, but orders are fine. Maybe you need customer data back but not the 80+ log and history collections that would overwrite recent entries.

MongoDB's documentation says `--nsInclude` should filter your restore to only the specified collections. And it does, **if you're restoring from a directory dump** (individual `.bson` files per collection).

But if you backed up with `--archive` and `--gzip` (which is what most production pipelines use, because who wants thousands of individual BSON files when you can have a single compressed stream to S3?), `--nsInclude` **silently restores everything anyway.**

I discovered this the hard way. I ran something like:

```
# What SHOULD work according to the docs
mongorestore \
  --gzip --archive=latest.dump.gz \
  --nsInclude="mydb.products" \
  --nsInclude="mydb.orders"
```

Expected: restore only products and orders.

Actual: mongorestore went ahead and restored every collection in the archive. All 130+ of them.

I thought I was doing something wrong. I couldn't find any documentation explaining this behavior. Then I found a MongoDB Community Forums thread from August 2024 where a user reported the exact same thing, backups created with `mongodump --archive --gzip`, and `--nsInclude` ignored during restore. A MongoDB community moderator tested it and confirmed: even using `--nsFrom` / `--nsTo` to target a single collection from an archive, mongorestore **still tries to restore the other collections**, generating duplicate key errors on everything it wasn't supposed to touch.

There's even a MongoDB JIRA ticket (TOOLS-2023) acknowledging that the documentation around gzip is confusing and that "selectivity logic" needs improvement. That ticket has been open for **over six years**.

Why it happens: A directory dump has individual `.bson` files per collection, mongorestore can simply skip the files it doesn't need. But an `--archive` stream is a single multiplexed binary. Mongorestore has to read through the entire stream sequentially, it can't seek. The namespace filtering doesn't reliably prevent restoration of non-matching collections when the source is a gzipped archive.

The docs say `--nsInclude` works with `--archive`. In practice, with `--gzip --archive`, it doesn't.

IT GETS WORSE: YOU CAN'T INSPECT THE ARCHIVE

Here's the part that made the whole experience truly painful.

When `--nsInclude` failed and I realized I needed to use `--nsExclude` for every collection I didn't want restored, my next thought was: let me list what's in the archive so I can build the exclude list.

You can't.

There is no built-in command to list the collections inside a `--gzip --archive` file. MongoDB provides no `--list` flag, no `--inspect` option, no way to peek inside. The `--dryRun` flag exists, but looking at the source code, it completes before the archive is actually demuxed, it doesn't enumerate what's inside.

A directory dump? Easy, just `ls` the folder. But a gzip archive is an opaque binary blob. You either restore it or you don't. There's nothing in between.

So I had to build my exclude list from memory and from querying the live database with `show collections`. For a database with 130+ collections that had grown organically over a decade, history tables, audit logs, staging collections, error archives, metrics aggregates, and half-forgotten import tables, this was not a five-minute exercise.

This is why the backup script saves a collection inventory file. Every backup gets a `.collections.txt` alongside its `.dump.gz`. When you need to do a selective restore six months later, you don't have to guess what's inside the archive. You just read the file.

THE WORKAROUND: `--NSEXCLUDE` EVERYTHING YOU DON'T WANT

Since `--nsInclude` can't be trusted with gzipped archive restores, the only reliable approach is the inverse: explicitly exclude every collection you don't want restored.

On my e-commerce platform with 34 sites, a production restore command had **130+ `--nsExclude` flags**. Every history table. Every log collection. Every analytics aggregate. Every staging table. Every error archive. The core business data that actually needed restoring was maybe 15 collections out of 130+.

Building that command by hand is error-prone and slow, exactly what you don't want during an incident. So I wrote a script that generates the restore command from the collection inventory file:

```
#!/bin/bash
set -e

# ============================================================
# MongoDB Selective Restore Command Builder
# ============================================================
# Generates mongorestore commands using the collection inventory
# file created by the backup script.
#
# Why this exists:
#   - --nsInclude doesn't work reliably with --gzip --archive
#   - You can't list collections inside a gzip archive
#   - Building 130+ --nsExclude flags by hand at 2 AM is a mistake
#
# Usage:
#   ./mongo_restore_builder.sh <collections_file> <mode> [collections...]
#
# Modes:
#   include  - Restore ONLY the listed collections
#   exclude  - Restore everything EXCEPT the listed collections
#   tier1    - Restore only Tier 1 (critical) collections
#
# Examples:
#   ./mongo_restore_builder.sh latest.collections.txt include products orders
#   ./mongo_restore_builder.sh latest.collections.txt exclude sessions email_lo
#   ./mongo_restore_builder.sh latest.collections.txt tier1
# ============================================================

# --- Configuration ---
MONGO_URI="mongodb+srv://user:pass@cluster.mongodb.net"
MONGO_DB="products"
ARCHIVE_PATH="/data/temp/latest.dump.gz"

# --- Tier 1: Critical business data ---
# Edit this list for your database
TIER1_COLLECTIONS=(
  "orders"
  "customers"
  "products"
  "inventory"
  "pricing"
  "webUsers"
  "employees"
  "categories"
  "brands"
  "pages"
```

```
  "systemTemplates"
)

# --- Parse arguments ---
COLLECTIONS_FILE="$1"
MODE="$2"
shift 2 2>/dev/null || true
SELECTED_COLLECTIONS=("$@")

if [ ! -f "$COLLECTIONS_FILE" ]; then
  echo "Error: Collections file not found: $COLLECTIONS_FILE"
  echo "Download it: aws s3 cp s3://your-bucket/mongo_backups/products/latest.c
  exit 1
fi

if [ -z "$MODE" ]; then
  echo "Usage: $0 <collections_file> <include|exclude|tier1> [collections...]"
  echo ""
  echo "Collections in this backup ($(wc -l < "$COLLECTIONS_FILE") total):"
  cat "$COLLECTIONS_FILE"
  exit 0
fi

# --- Read all collections ---
ALL_COLLECTIONS=()
while IFS= read -r line; do
  [ -n "$line" ] && ALL_COLLECTIONS+=("$line")
done < "$COLLECTIONS_FILE"

# --- Build exclude list based on mode ---
EXCLUDE_LIST=()

case "$MODE" in
  include)
    # Restore ONLY these collections → exclude everything else
    for col in "${ALL_COLLECTIONS[@]}"; do
      SKIP=false
      for selected in "${SELECTED_COLLECTIONS[@]}"; do
        [ "$col" = "$selected" ] && SKIP=true && break
      done
      [ "$SKIP" = false ] && EXCLUDE_LIST+=("$col")
    done
    ;;
  exclude)
    # Exclude these collections → restore everything else
    EXCLUDE_LIST=("${SELECTED_COLLECTIONS[@]}")
    ;;
```

```
  tier1)
    # Restore only Tier 1 → exclude everything not in TIER1_COLLECTIONS
    for col in "${ALL_COLLECTIONS[@]}"; do
      SKIP=false
      for tier1 in "${TIER1_COLLECTIONS[@]}"; do
        [ "$col" = "$tier1" ] && SKIP=true && break
      done
      [ "$SKIP" = false ] && EXCLUDE_LIST+=("$col")
    done
    ;;
esac

# --- Generate the command ---
echo "mongorestore \\"
echo "  --uri=\"${MONGO_URI}\" \\"
echo "  --gzip --archive=${ARCHIVE_PATH} \\"

for i in "${!EXCLUDE_LIST[@]}"; do
  if [ $i -eq $(( ${#EXCLUDE_LIST[@]} - 1 )) ]; then
    echo "  --nsExclude=\"${MONGO_DB}.${EXCLUDE_LIST[$i]}\""
  else
    echo "  --nsExclude=\"${MONGO_DB}.${EXCLUDE_LIST[$i]}\" \\"
  fi
done

echo ""
echo "# Excluding ${#EXCLUDE_LIST[@]} of ${#ALL_COLLECTIONS[@]} collections"
```

Now instead of building a 130-line command under pressure, it's:

```
# Download the collection inventory
aws s3 cp s3://your-bucket/mongo_backups/products/latest.collections.txt .

# "What's in this backup?"
./mongo_restore_builder.sh latest.collections.txt
# → prints all 130+ collection names

# "Restore only the products collection"
./mongo_restore_builder.sh latest.collections.txt include products

# "Restore only critical business data"
./mongo_restore_builder.sh latest.collections.txt tier1

# "Restore everything except sessions and logs"
./mongo_restore_builder.sh latest.collections.txt exclude sessions email_log br
```

The `tier1` mode is the one you'll use most. It maps to the collection tiering strategy below.

THE COLLECTION TIERING STRATEGY THAT SAVES YOU AT 2 AM

I tier every collection in the database:

Tier 1, Critical business data. Orders, customers, products, inventory, pricing. Always restore these. If you lose them, the business stops.

Tier 2, Regenerable. Sessions, caches, search indexes, login tokens. Never restore these. They rebuild themselves. Restoring old sessions would actually be worse than having none, you'd be logging people into stale states.

Tier 3, Historical/analytical. Audit logs, history tables, analytics aggregates, import logs, error archives. Restore only if specifically needed. These are the 100+ collections that make up the bulk of your exclude list.

The `TIER1_COLLECTIONS` array in the restore builder script is your runbook. Edit it once, and every restore after that is a single command. When the moment comes, you want to run a command, not write one.

THE SELF-HEALING TEST NOBODY RUNS

Everyone talks about replica set failover. Almost nobody actually tests it.

I've deliberately destroyed replica set members multiple times, not because something broke, but because I wanted to know exactly what happens when something does.

The experiment: Take a secondary offline. Delete the entire data directory. Every collection, every index, every byte of data. Then start the `mongod` process and let it rejoin the replica set.

What MongoDB does next is genuinely impressive to watch. The rejoining member detects it has no data, triggers an initial sync from the primary, and rebuilds itself, cloning every collection, rebuilding every index in parallel, then applying buffered oplog entries to catch up to the current state. All automatic. No manual intervention.

And you can watch the entire process in real time:

```
# Connect to the rebuilding member
mongosh --host rebuilding-member:27017

# Watch the replica set status,  the member will show as STARTUP2 during sync
rs.status().members.forEach(m => {
  print(`${m.name}: ${m.stateStr} | health: ${m.health}`)
})

# Monitor initial sync progress in detail
# (only available while the member is in STARTUP2 state)
db.adminCommand({ replSetGetStatus: 1, initialSync: 1 }).initialSyncStatus

# This returns:
# - totalInitialSyncElapsedMillis (how long it's been syncing)
# - remainingInitialSyncEstimatedMillis (estimated time left)
# - approxTotalDataSize (total data to copy)
# - approxTotalBytesCopied (progress so far)
# - databases,  per-database breakdown of collections being cloned

# Check replication lag once the member transitions to SECONDARY
rs.printSecondaryReplicationInfo()

# Watch the oplog catch-up in real time
rs.status().members.forEach(m => {
  if (m.stateStr === "SECONDARY") {
    const lag = (rs.status().members.find(p => p.stateStr === "PRIMARY").optime
                - m.optimeDate) / 1000
    print(`${m.name}: ${lag}s behind primary`)
  }
})
```

On my production dataset, watching the `approxTotalBytesCopied` tick upward against the `approxTotalDataSize` while indexes rebuild in parallel, it's like watching a surgeon work. Fast, methodical, and the member transitions from `STARTUP2` to `SECONDARY` in far less time than you'd expect for a full dataset rebuild.

Then I got mean.

I killed the member again. Mid-rebuild. While it was still in `STARTUP2`, actively cloning data from the primary. Pulled the plug, nuked the data directory a second time, and started it back up.

MongoDB didn't flinch. The member detected the failed initial sync, reset, and started the process over from scratch. No corruption. No confused state. No manual cleanup needed. It just started syncing again as if nothing happened. The `failedInitialSyncAttempts` counter incremented by one, and the rebuild continued.

I did this three times in a row on the same member. Delete everything, start, kill mid-sync, delete everything, start again. Every time, the replica set absorbed the disruption and the member eventually rebuilt itself to a fully consistent state.

The point isn't that MongoDB can do this. It's that you should verify it can do this with your data, your network, and your topology before you need it to. Run this test in staging. Watch the shell output. Know exactly how long your replica set takes to rebuild a member from zero. That number matters when you're on a call at 2 AM deciding whether to wait for self-healing or intervene with a manual restore from backup.

WRITE CONCERN: THE BACKUP DECISION YOU'RE MAKING WITHOUT REALIZING IT

Your write concern setting directly determines whether your replica set is a backup or just a mirror.

`w: 1`, Write acknowledged by the primary only. If the primary dies before replicating, that write is gone. You have no backup of it. It never existed on any other node.

`w: "majority"`, Write acknowledged by the majority of replica set members. The data exists on multiple nodes before your application gets the OK. This is an actual backup.

`w: 0`, Fire and forget. No acknowledgment at all. Only use this for data you genuinely don't care about losing.

The performance difference is real. Especially cross-region, `w: "majority"` means the write has to cross the Atlantic before acknowledging. That's roughly 100ms added to every write.

So I split by data criticality:

- **Orders, customers, inventory:** `w: "majority"`, can't lose it
- **Sessions, caches:** `w: 1`, regenerated easily
- **Analytics, telemetry:** `w: 1`, losing a data point doesn't matter

That single decision, matching write concern to data criticality instead of applying one setting globally, was probably the most impactful performance optimization we made across the entire platform. And it's a backup decision disguised as a performance decision.

THE MISTAKES THAT TAUGHT ME THESE LESSONS

Year 2: The WiredTiger memory lesson. MongoDB's WiredTiger engine defaults to 50% of available RAM. On a 16GB EC2 m5d.xlarge, that's 8GB claimed before your application gets anything. We were also running Elasticsearch on the same instances, which also wants 50% for JVM heap. During a traffic spike, our Node.js workers got OOM-killed. MongoDB and Elasticsearch were both doing exactly what they were configured to do. We just hadn't configured them. Now I cap WiredTiger at 40% of available memory on every deployment, no exceptions.

Year 4: The migration that locked the primary. Ran a schema migration on the primary during business hours. Write lock cascaded to a 30-second pause across 34 websites. Now all migrations run on a hidden secondary first, validated, then applied to primary during maintenance windows.

Year 5: The OS update that broke replication. A routine `apt upgrade` pulled a new OpenSSL version that changed TLS behavior. Replica set members couldn't authenticate. The fix: pin MongoDB and all its dependencies. Every MongoDB version

change is a deliberate, tested event. Never a side effect of maintenance.

Year 7: The disk that filled up. Primary went read-only because I didn't set up log rotation for MongoDB's diagnostic logs. Not the data. Not the oplog. The diagnostic logs. Now I use `systemLog.logRotate: rename` with a cron job and monitor disk usage with alerts at 80%.

Year 9: The major version upgrade. Upgraded without reading the compatibility notes. A deprecated aggregation operator I used heavily had been removed. Rollback took 2 hours. Now I test every major version upgrade against a clone of production data before touching the real thing.

None of these caused data loss. The replica set and the backup pipeline protected me every time. That's the entire point.