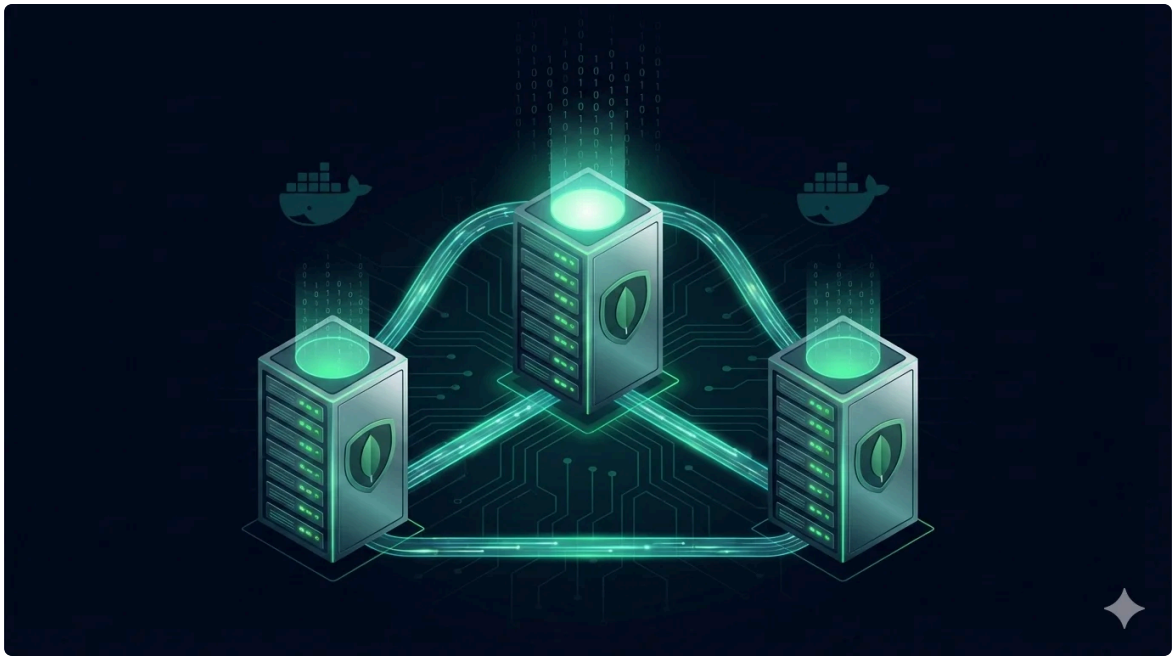


MongoDB Replica Sets Guide

Production-Ready on Ubuntu

DECEMBER 15, 2025

THEDECIPHERIST.COM

TABLE OF CONTENTS

REDDIT TL;DR

Setting up MongoDB in production? Here's the 2-minute version:

🔧 Setup Essentials:

- Use **3 nodes minimum** (1 primary, 2 secondaries) for quorum
- **XFS filesystem** - WiredTiger performs significantly better on XFS than ext4
- **DNS hostnames required** - MongoDB 5.0+ fails startup with IP-only configs
- Use `mongosh` not `mongo` (deprecated/removed in 6.0+)
- Use `--tls` not `--ssl` (deprecated since 4.2)
- Use `gpg --dearmor` not `apt-key add` (deprecated)

⚡ Performance Quick Wins:

- Disable Transparent Huge Pages (THP) - causes serious latency spikes
- Set `vm.swappiness=1`
- Set WiredTiger cache to ~50% of RAM minus 1GB
- Use `$match` FIRST in aggregation pipelines (uses indexes)
- Follow ESR rule for compound indexes: **E**quality → **S**ort → **R**ange

🔒 Security Non-Negotiables:

- **MongoDB should be completely unreachable from the public internet** — not just "protected", but invisible
- Public users → Reverse proxy (nginx) → App server → MongoDB (internal network only)
- Use internal DNS that only resolves within your private network
- Enable authentication with keyfile
- Use TLS for all connections
- Never expose port 27017 to the internet
- Use `w: "majority"` write concern for critical data
- (Atlas) Whitelist only your app server IPs, never `0.0.0.0/0`

📊 Debugging Slow Queries:

```
// Enable profiler for queries >100ms (disable when done!)
db.setProfilingLevel(1, { slowms: 100 })

// Check slow queries
db.system.profile.find().sort({ ts: -1 }).limit(10)

// Enable verbose command logging
db.setLogLevel(1, "command")
```

⚠️ **Profiler Warning:** Level 2 profiling can KILL production performance. Use level 1 with high `slowms`, keep sessions short, always disable when done.

🔗 Connection Pooling:

```
// Always configure pool settings explicitly
"mongodb://.../?maxPoolSize=100&minPoolSize=10&retryWrites=true&w=majority"
```

💾 Backup Reality Check:

- `mongodump` is fine for <100GB
- For larger DBs, use filesystem snapshots or Percona Backup
- **Always test restores** - untested backups aren't backups

💰 Atlas vs Self-Hosted:

- Atlas wins under ~$1,500/month (when you factor engineering time)
- Self-host at $2,000+/month Atlas spend with dedicated ops resources
- Never run MongoDB on ECS/Fargate - use EC2 with persistent storage

📐 Schema Design Rules:

- Embed data accessed together (orders + line items)
- Reference unbounded/large data (user → posts)
- Max document size is 16MB, but aim for <1MB
- Never use unbounded arrays that grow forever

🚨 Test Your Failover!

```
rs.stepDown(60)  // Force election - do this regularly!
```

🐳 **Docker Deployment Rules:**

- Use bind mounts, NOT anonymous volumes (data loss risk!)
- One MongoDB container per physical host (use placement constraints)
- Use `mode: host` for ports, NOT ingress (breaks replica set!)
- Use Docker secrets for passwords, never plain text in compose
- Container hostnames in rs.initiate(), NOT localhost
- Set WiredTiger cache = 50% of container memory - 1GB

Full guide covers: DNS setup, OS tuning, TLS certs, backup scripts, aggregation, indexing, profiling risks, transactions, monitoring/alerting, connection pooling, schema design, disaster recovery, **and complete Docker Swarm deployment with best practices**.

TABLE OF CONTENTS

PART 1: WHY REPLICA SETS?

If you're running MongoDB in production without a replica set, you're playing with fire. Here's what you get:

- **High Availability** - Automatic failover if your primary goes down
- **Data Redundancy** - Your data exists on multiple servers
- **Read Scaling** - Distribute read operations across secondaries
- **Zero-Downtime Maintenance** - Rolling upgrades and maintenance
- **ACID Transactions** - Multi-document transactions require replica sets

The minimum recommended setup is 3 nodes: 1 primary and 2 secondaries. This allows the cluster to maintain quorum even if one node fails.

What's New in MongoDB 8.0?

MongoDB 8.0 (released October 2024) brings significant improvements:

- **36% faster reads** and **59% higher throughput** for updates
- Improved horizontal scaling
- Enhanced Queryable Encryption with range queries
- Better performance across the board

PART 2: ATLAS VS SELF-HOSTED - WHEN TO CHOOSE WHAT

Before diving into self-hosted setup, let's address the elephant in the room: **Should you even self-host?**

PART 2.5: AUTOMATED INSTALLATION SCRIPT

Want to skip the manual steps? Download our production-ready installation script that automates everything in this guide.

📥 Download All Files

All scripts and configuration files are available for download:

FILE	DESCRIPTION	DOWNLOAD
`mongodb-install.sh`	Automated bare-metal installation script	View/Download
`docker-compose.yml`	Production Docker Swarm deployment	View/Download
`docker-compose.dev.yml`	Development single-host Docker setup	View/Download
`deploy-mongodb-swarm.sh`	Docker Swarm automation script	View/Download
`mongod.conf`	Optimized MongoDB configuration	Embedded in scripts

Quick download (copy-paste ready):

```
# Option 1: Create files directory
mkdir -p mongodb-setup && cd mongodb-setup

# Option 2: If hosted on GitHub (replace with your repo)
# git clone https://github.com/yourusername/mongodb-production-guide.git

# Option 3: Copy scripts directly from this guide (scroll down for full content
```

What the Script Does

✅ Configures hostname and `/etc/hosts`
✅ Formats data drive with XFS (optional)
✅ Applies all OS tuning (THP, swappiness, file limits, read-ahead)
✅ Installs MongoDB 8.0 using modern GPG keyring method
✅ Creates optimized `mongod.conf`
✅ Generates replica set keyfile
✅ Sets up log rotation
✅ Creates backup script template
✅ Creates health check script
✅ Optionally initializes replica set

Download and Usage

```
# Create a directory for MongoDB setup files
mkdir -p mongodb-setup && cd mongodb-setup

# Create the installation script (copy content from "The Complete Script" secti
nano mongodb-install.sh

# Make executable
chmod +x mongodb-install.sh

# Edit configuration section at the top of the script
nano mongodb-install.sh

# Run with sudo
sudo ./mongodb-install.sh
```

Configuration Variables

Edit these variables at the top of the script before running:

```
# Node Configuration
NODE_HOSTNAME="mongodb1.yourdomain.com"    # This node's FQDN
NODE_IP="10.10.1.122"                       # This node's private IP
REPLICA_SET_NAME="rs0"                      # Replica set name

# Other Replica Set Members
OTHER_NODES=(
    "10.10.1.175 mongodb2.yourdomain.com mongodb2"
    "10.10.1.136 mongodb3.yourdomain.com mongodb3"
)

# Data Drive (set to "" to skip formatting)
DATA_DRIVE="/dev/nvme1n1"
DATA_PATH="/data/mongodb"

# MongoDB Settings
WIREDTIGER_CACHE_GB="2"                     # 50% of RAM - 1GB

# Set these only on the PRIMARY node after all nodes are installed
INIT_REPLICA_SET="false"
ADMIN_PASSWORD=""                           # Set to create admin user
```

Multi-Node Deployment Steps

Step 1: Run on ALL nodes (with `INIT_REPLICA_SET=false`)

```
# On mongodb1, mongodb2, mongodb3
sudo ./mongodb-install.sh
```

Step 2: Copy keyfile to all nodes

```
# From mongodb1
scp /keys/mongodb.key user@mongodb2:/keys/mongodb.key
scp /keys/mongodb.key user@mongodb3:/keys/mongodb.key

# Fix permissions on each node
ssh user@mongodb2 'sudo chown mongodb:mongodb /keys/mongodb.key && sudo chmod 4
ssh user@mongodb3 'sudo chown mongodb:mongodb /keys/mongodb.key && sudo chmod 4
```

Step 3: Initialize replica set (on primary only)

```
# On mongodb1
mongosh --eval '
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongodb1.yourdomain.com:27017", priority: 2 },
    { _id: 1, host: "mongodb2.yourdomain.com:27017", priority: 1 },
    { _id: 2, host: "mongodb3.yourdomain.com:27017", priority: 1 }
  ]
})'
```

Step 4: Create admin user

```
mongosh --eval '
use admin
db.createUser({
  user: "adminUser",
  pwd: "YourStrongPassword123!",
  roles: [{ role: "root", db: "admin" }]
})'
```

Step 5: Enable authentication on ALL nodes

```
# Edit /etc/mongod.conf - uncomment security section:
security:
  authorization: enabled
  keyFile: /keys/mongodb.key

# Restart MongoDB
sudo systemctl restart mongod
```

Step 6: Verify

```
# Test connection
mongosh "mongodb://mongodb1.yourdomain.com:27017,mongodb2.yourdomain.com:27017,
  -u adminUser -p

# Run health check
/opt/mongodb/scripts/health-check.sh
```

The Complete Script

▶ Click to expand the full installation script (~500 lines)

PART 2.6: DOCKER SWARM DEPLOYMENT

Prefer containers? Here's a production-ready Docker Swarm deployment for MongoDB replica sets.

⚠️ Docker Best Practices for MongoDB Replica Sets

Running MongoDB in Docker requires careful attention to several critical areas. **Ignoring these can lead to data loss, poor performance, or cluster instability.**

1. Persistent Storage (CRITICAL)

The Problem: Docker containers are ephemeral. Without proper volume configuration, your data disappears when containers restart.

```
# ❌ BAD: Anonymous volume (data loss risk on container recreation)
volumes:
  - /data/db

# ❌ BAD: Named volume without host binding (hard to backup, inspect)
volumes:
  - mongo-data:/data/db

# ✅ GOOD: Bind mount to host directory (full control, easy backup)
volumes:
  - /data/mongodb/node1:/data/db

# ✅ BEST: Bind mount with explicit driver options (Swarm)
volumes:
  mongo1-data:
    driver: local
    driver_opts:
      type: none
      o: bind
      device: /data/mongodb/node1
```

Best Practices:

- Always use bind mounts to host directories for production
- Create data directories BEFORE deploying: `mkdir -p /data/mongodb && chown 999:999 /data/mongodb`
- Use XFS filesystem on the host for best performance
- Never use NFS or network storage for MongoDB data (latency kills performance)

2. Container Placement (CRITICAL for Swarm)

The Problem: If all 3 replica set members land on the same host, you have no fault tolerance.

```
# ❌ BAD: No placement constraints (all containers might run on same node)
deploy:
  replicas: 1

# ✅ GOOD: Pin each MongoDB instance to a specific node
deploy:
  placement:
    constraints:
      - node.labels.mongo.replica == 1
```

Setup node labels:

```
docker node update --label-add mongo.replica=1 node1-hostname
docker node update --label-add mongo.replica=2 node2-hostname
docker node update --label-add mongo.replica=3 node3-hostname
```

Best Practices:

- One MongoDB container per physical/virtual host
- Label nodes and use placement constraints
- For single-node testing only: accept that you have no real HA

3. Networking

The Problem: MongoDB replica set members must be able to reach each other by hostname, and clients need consistent connection strings.

```
# ❌ BAD: Default bridge network (containers can't resolve each other)
networks:
  - default

# ✅ GOOD: Custom bridge network (development)
networks:
  mongo-cluster:
    driver: bridge

# ✅ GOOD: Overlay network (Swarm production)
networks:
  mongo-net:
    driver: overlay
    attachable: true
```

Best Practices:

- Use custom networks, never the default bridge
- Container hostnames become DNS names within the network
- Use `hostname:` in compose to set predictable names (mongo1, mongo2, mongo3)
- For external access, use `mode: host` for ports (not ingress/load-balanced)

```
# ❌ BAD: Ingress mode load-balances across all replicas (breaks replica set!)
ports:
  - "27017:27017"

# ✅ GOOD: Host mode exposes port on the specific node
ports:
  - target: 27017
    published: 27017
    mode: host
```

4. Security

The Problem: Default MongoDB has no authentication. In Docker, secrets management is different.

```
# ❌ BAD: Password in plain text in compose file
environment:
  - MONGO_INITDB_ROOT_PASSWORD=mysecretpassword

# ✅ GOOD: Use Docker secrets (Swarm)
secrets:
  - mongodb-keyfile
  - mongodb-root-password

# Create secrets:
# openssl rand -base64 756 > keyfile && docker secret create mongodb-keyfile ke
# echo -n "password" | docker secret create mongodb-root-password -
```

Best Practices:

- Always enable authentication (`--keyFile` for replica sets)
- Use Docker secrets for passwords and keyfiles
- Keyfile must have restricted permissions (handled automatically with secrets)
- Never expose MongoDB ports (27017) to the internet

5. Resource Limits

The Problem: Without limits, MongoDB can consume all host memory and starve other containers.

```
# ❌ BAD: No resource limits
deploy:
  replicas: 1

# ✅ GOOD: Explicit memory limits
deploy:
  resources:
    limits:
      memory: 4G      # Hard cap
    reservations:
      memory: 2G      # Guaranteed minimum
```

WiredTiger Cache Sizing:

```
# Formula: 50% of container memory limit - 1GB
# For 4GB container limit: (4 * 0.5) - 1 = 1GB cache

mongod --wiredTigerCacheSizeGB 1
```

Best Practices:

- Always set memory limits
- Set WiredTiger cache to ~50% of container memory minus 1GB
- Leave headroom for connections, aggregations, and sorting

6. Health Checks

The Problem: Docker needs to know if MongoDB is actually healthy, not just running.

```
# ❌ BAD: No health check (Docker thinks container is healthy if process runs)
# (no healthcheck defined)

# ✅ GOOD: Actual MongoDB health check
healthcheck:
  test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 60s  # Give MongoDB time to start
```

Best Practices:

- Always define health checks
- Use `start_period` to avoid false failures during startup
- Check actual MongoDB responsiveness, not just process existence

7. Replica Set Initialization

The Problem: Containers start independently. The replica set must be manually initialized after all members are running.

Correct initialization order:

1. Start all 3 MongoDB containers

2. Wait for them to be healthy (30-60 seconds)
3. Connect to ONE container and run `rs.initiate()`
4. Create admin user
5. Enable authentication (if not already enabled)

```
# Wait for all containers to be ready
sleep 30

# Initialize from mongo1
docker exec mongo1 mongosh --eval '
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo1:27017", priority: 2 },
    { _id: 1, host: "mongo2:27017", priority: 1 },
    { _id: 2, host: "mongo3:27017", priority: 1 }
  ]
})'

# IMPORTANT: Use container hostnames (mongo1, mongo2, mongo3), not localhost!
```

Common Mistakes:

- ❌ Using `localhost:27017` in replica set config (other members can't reach it)
- ❌ Initializing before all members are running
- ❌ Not waiting for primary election before creating users

8. Logging and Debugging

```
# View logs
docker logs mongo1 -f
docker service logs mongodb_mongo1 -f  # Swarm

# Shell access
docker exec -it mongo1 mongosh

# Check replica set status
docker exec mongo1 mongosh --eval "rs.status()"

# Check if primary
docker exec mongo1 mongosh --eval "rs.isMaster().ismaster"
```

9. Backup Considerations

The Problem: Backing up containerized MongoDB requires accessing the data correctly.

```
# Option 1: Run mongodump from inside a container
docker exec mongo1 mongodump --archive=/data/db/backup.gz --gzip
docker cp mongo1:/data/db/backup.gz ./backup.gz

# Option 2: Run mongodump from host pointing to container
mongodump --host localhost:27017 --gzip --archive=backup.gz

# Option 3: Backup the bind-mounted directory (stop container first for consist
docker stop mongo1
tar -czvf backup.tar.gz /data/mongodb/node1
docker start mongo1
```

10. What NOT to Do

❌ DON'T	✅ DO INSTEAD
Use anonymous volumes	Use bind mounts to host directories
Run all replicas on one host (production)	Use placement constraints
Expose ports via ingress/load balancer	Use `mode: host` for ports
Put passwords in compose files	Use Docker secrets
Skip health checks	Always define health checks
Use `localhost` in replica set config	Use container hostnames
Forget to initialize the replica set	Script the initialization
Ignore WiredTiger cache sizing	Set cache based on container memory

Why Docker Swarm for MongoDB?

PROS	CONS
✅ Consistent deployments	⚠️ Networking complexity
✅ Easy scaling of app layer	⚠️ Persistent storage challenges
✅ Service discovery built-in	⚠️ Less control over OS tuning
✅ Rolling updates	⚠️ Additional abstraction layer
✅ Secrets management	⚠️ Debugging is harder

Quick Start (Development - Single Host)

For local development/testing:

```
# Create directory
mkdir mongodb-docker && cd mongodb-docker

# Generate keyfile
openssl rand -base64 756 > mongodb-keyfile
chmod 600 mongodb-keyfile

# Create docker-compose.dev.yml (content below)
# Then start containers
docker-compose -f docker-compose.dev.yml up -d

# Wait for startup, then initialize replica set
sleep 10
docker exec mongo1 mongosh --eval '
  rs.initiate({
    _id: "rs0",
    members: [
      { _id: 0, host: "mongo1:27017", priority: 2 },
      { _id: 1, host: "mongo2:27017", priority: 1 },
      { _id: 2, host: "mongo3:27017", priority: 1 }
    ]
  })'

# Create admin user
sleep 5
docker exec mongo1 mongosh --eval '
  use admin
  db.createUser({
    user: "admin",
    pwd: "password123",
    roles: ["root"]
  })'
```

Connection string:

```
mongodb://admin:password123@localhost:27017,localhost:27018,localhost:27019/?re
```

docker-compose.dev.yml (Single Host)

```
version: "3.8"

services:
  mongo1:
    image: mongo:8.0
    container_name: mongo1
    hostname: mongo1
    ports:
      - "27017:27017"
    volumes:
      - mongo1-data:/data/db
      - ./mongodb-keyfile:/etc/mongodb-keyfile:ro
    command: >
      mongod
      --replSet rs0
      --bind_ip_all
      --keyFile /etc/mongodb-keyfile
      --wiredTigerCacheSizeGB 0.5
    networks:
      - mongo-cluster
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 30s

  mongo2:
    image: mongo:8.0
    container_name: mongo2
    hostname: mongo2
    ports:
      - "27018:27017"
    volumes:
      - mongo2-data:/data/db
      - ./mongodb-keyfile:/etc/mongodb-keyfile:ro
    command: >
      mongod
      --replSet rs0
      --bind_ip_all
      --keyFile /etc/mongodb-keyfile
      --wiredTigerCacheSizeGB 0.5
    networks:
      - mongo-cluster
    depends_on:
      - mongo1
```

```
  mongo3:
    image: mongo:8.0
    container_name: mongo3
    hostname: mongo3
    ports:
      - "27019:27017"
    volumes:
      - mongo3-data:/data/db
      - ./mongodb-keyfile:/etc/mongodb-keyfile:ro
    command: >
      mongod
      --replSet rs0
      --bind_ip_all
      --keyFile /etc/mongodb-keyfile
      --wiredTigerCacheSizeGB 0.5
    networks:
      - mongo-cluster
    depends_on:
      - mongo1

  # Optional: Web UI
  mongo-express:
    image: mongo-express:1.0
    container_name: mongo-express
    ports:
      - "8081:8081"
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=admin
      - ME_CONFIG_MONGODB_ADMINPASSWORD=password123
      - ME_CONFIG_MONGODB_URL=mongodb://admin:password123@mongo1:27017,mongo2:2
      - ME_CONFIG_BASICAUTH_USERNAME=admin
      - ME_CONFIG_BASICAUTH_PASSWORD=admin123
    networks:
      - mongo-cluster
    depends_on:
      - mongo1
      - mongo2
      - mongo3

networks:
  mongo-cluster:
    driver: bridge

volumes:
  mongo1-data:
```

```
  mongo2-data:
  mongo3-data:
```

Production Docker Swarm Deployment

For production with multiple physical nodes:

Step 1: Initialize Swarm and label nodes

```
# On manager node
docker swarm init

# Label nodes for MongoDB placement
docker node update --label-add mongo.replica=1 node1
docker node update --label-add mongo.replica=2 node2
docker node update --label-add mongo.replica=3 node3
```

Step 2: Create network and secrets

```
# Create overlay network
docker network create --driver overlay --attachable mongo-net

# Generate and create keyfile secret
openssl rand -base64 756 > mongodb-keyfile
docker secret create mongodb-keyfile mongodb-keyfile

# Create password secret
echo -n "YourSecurePassword123!" | docker secret create mongodb-root-password -
```

Step 3: Create data directories (on each node)

```
# MongoDB runs as UID 999 in container
mkdir -p /data/mongodb
chown 999:999 /data/mongodb
```

Step 4: Deploy stack

```
docker stack deploy -c docker-compose.yml mongodb
```

Step 5: Initialize replica set

```
# Wait for containers to start
sleep 30

docker exec $(docker ps -qf name=mongodb_mongo1) mongosh --eval '
  rs.initiate({
    _id: "rs0",
    members: [
      { _id: 0, host: "mongo1:27017", priority: 2 },
      { _id: 1, host: "mongo2:27017", priority: 1 },
      { _id: 2, host: "mongo3:27017", priority: 1 }
    ]
  })'
```

docker-compose.yml (Docker Swarm - Production)

▶ Click to expand full Swarm docker-compose.yml

Docker vs Bare Metal: When to Use What

USE DOCKER WHEN	USE BARE METAL WHEN
Development/testing	Maximum performance needed
Consistent environments	Fine-grained OS tuning required
CI/CD pipelines	Very large datasets (TB+)
Microservices architecture	Compliance requires no containers
Quick prototyping	Existing bare metal infrastructure

⚠️ **Production Warning:** While Docker Swarm works for MongoDB, be aware:

- Persistent storage is more complex than bare metal
- Network performance may be slightly lower
- OS tuning options are limited inside containers
- Debugging requires container knowledge

For mission-critical production workloads with high performance requirements, consider bare metal or VMs with the installation script from Part 2.5.

PART 3: ATLAS VS SELF-HOSTED - WHEN TO CHOOSE WHAT

The Honest Truth

For most teams, **MongoDB Atlas is the right choice** until you hit certain thresholds:

FACTOR	ATLAS WINS	SELF-HOSTED WINS
Time to production	Minutes	Days/weeks
Operational overhead	Near zero	Significant
Backups	Automatic, point-in-time	Manual setup required
Cost at small scale	Lower TCO	Higher TCO
Cost at large scale	Can become expensive	Potentially 50-70% savings

The Break-Even Analysis

```
Atlas M50 (3-node): ~$1,440/month

Self-hosted equivalent on AWS:
  3x r6g.xlarge: ~$441/month
  + Storage, backups, transfer: ~$120/month
  = ~$560/month infrastructure

BUT factor in:
  - Engineering setup: 40-80 hours
  - Monthly maintenance: 4-8 hours
  - On-call burden

Break-even: ~$2,000-3,000/month Atlas spend
```

Stay on Atlas when: Small team, under $1,500/month, no DBA expertise **Self-host when:** Over $2,000/month Atlas, dedicated ops resources, compliance requirements

---

PART 3: INITIAL SERVER SETUP

Prerequisites

- 3 Ubuntu servers (22.04 LTS or 24.04 LTS) - **Ubuntu 18.04 is no longer supported**
- Root/sudo access on all servers
- Private network connectivity between nodes
- A dedicated data drive (separate from OS) on each node

Network Planning

NODE	PRIVATE IP	HOSTNAME
Primary	10.10.1.122	mongodb1.yourdomain.com
Secondary 1	10.10.1.175	mongodb2.yourdomain.com
Secondary 2	10.10.1.136	mongodb3.yourdomain.com

⚠️ **Important:** Starting in MongoDB 5.0, nodes configured with only an IP address will fail startup validation. **Always use DNS hostnames** for replica set members.

Step 3.1: Configure Hostnames (All Nodes)

```
# On mongodb1
sudo hostnamectl set-hostname mongodb1.yourdomain.com

# On mongodb2
sudo hostnamectl set-hostname mongodb2.yourdomain.com

# On mongodb3
sudo hostnamectl set-hostname mongodb3.yourdomain.com
```

Step 3.2: Configure /etc/hosts (All Nodes)

```
sudo nano /etc/hosts
```

Add:

```
10.10.1.122 mongodb1.yourdomain.com mongodb1
10.10.1.175 mongodb2.yourdomain.com mongodb2
10.10.1.136 mongodb3.yourdomain.com mongodb3
```

Step 3.3: Update the System

```
sudo apt-get update && sudo apt-get upgrade -y
```

PART 4: FILESYSTEM SETUP

This is where most guides fail you. MongoDB with WiredTiger storage engine performs significantly better on XFS filesystem.

Step 4.1: Install XFS Tools

```
sudo apt-get install xfsprogs -y
```

Step 4.2: Format the Data Drive

⚠️ **WARNING:** This will destroy all data on the drive!

```
# Check your drives first
lsblk

# Format with XFS (replace /dev/nvme1n1 with your drive)
sudo mkfs.xfs /dev/nvme1n1
```

Step 4.3: Mount the Drive

```
sudo mkdir /data
sudo mount /dev/nvme1n1 /data/
df -T  # Verify it's mounted with xfs
```

Step 4.4: Configure Persistent Mount

```
# Get the UUID
sudo blkid /dev/nvme1n1

# Add to fstab
sudo nano /etc/fstab
```

Add (replace UUID):

```
UUID=your-uuid-here /data xfs defaults,noatime 1 1
```

Test:

```
sudo mount -a && df -T
```

PART 5: OS TUNING FOR MONGODB

Step 5.1: Increase File Descriptor Limits

```
sudo nano /etc/security/limits.conf
```

Add:

```
* soft nofile 64000
* hard nofile 64000
* soft nproc 32000
* hard nproc 32000
```

Step 5.2: Disable Transparent Huge Pages (THP)

THP causes serious performance problems for databases:

```
sudo nano /etc/init.d/disable-transparent-hugepages
```

Paste:

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          disable-transparent-hugepages
# Required-Start:    $local_fs
# Required-Stop:
# X-Start-Before:    mongod mongodb-mms-automation-agent
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Disable Linux transparent huge pages
### END INIT INFO

case $1 in
  start)
    if [ -d /sys/kernel/mm/transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/transparent_hugepage
    elif [ -d /sys/kernel/mm/redhat_transparent_hugepage ]; then
      thp_path=/sys/kernel/mm/redhat_transparent_hugepage
    else
      return 0
    fi

    echo 'never' > ${thp_path}/enabled
    echo 'never' > ${thp_path}/defrag

    unset thp_path
    ;;
esac
```

Enable:

```
sudo chmod 755 /etc/init.d/disable-transparent-hugepages
sudo update-rc.d disable-transparent-hugepages defaults
```

Step 5.3: Set Swappiness

```
sudo nano /etc/sysctl.conf
```

Add:

```
vm.swappiness=1
```

Step 5.4: Optimize Read-Ahead (EC2/Cloud)

```
sudo crontab -e
```

Add:

```
@reboot /sbin/blockdev --setra 32 /dev/nvme1n1
```

Reboot all nodes:

```
sudo reboot
```

PART 6: INSTALL MONGODB 8.0

Step 6.1: Import MongoDB GPG Key (Modern Method)

⚠️ **The old `apt-key add` method is deprecated!** Use the new keyring approach:

```
# Install required tools
sudo apt-get install gnupg curl -y

# Import key using the modern method
curl -fsSL https://www.mongodb.org/static/pgp/server-8.0.asc | \
  sudo gpg -o /usr/share/keyrings/mongodb-server-8.0.gpg --dearmor
```

Step 6.2: Add MongoDB Repository

For **Ubuntu 24.04 (Noble)**:

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-8.0.g
  sudo tee /etc/apt/sources.list.d/mongodb-org-8.0.list
```

For Ubuntu 22.04 (Jammy):

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-8.0.g
  sudo tee /etc/apt/sources.list.d/mongodb-org-8.0.list
```

Step 6.3: Install MongoDB

```
sudo apt-get update
sudo apt-get install -y mongodb-org
```

Step 6.4: Create Data Directory

```
sudo mkdir -p /data/mongodb
sudo chown -R mongodb:mongodb /data/mongodb
sudo chmod -R 775 /data/mongodb
```

PART 7: CONFIGURE MONGODB

Step 7.1: Edit MongoDB Configuration

```
sudo nano /etc/mongod.conf
```

Production-ready configuration:

```
# Storage
storage:
  dbPath: /data/mongodb
  journal:
    enabled: true
  wiredTiger:
    engineConfig:
      cacheSizeGB: 2  # Adjust: typically 50% of RAM minus 1GB

# Logging
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log

# Network - Use THIS node's private IP
net:
  port: 27017
  bindIp: 10.10.1.122

# Replication
replication:
  replSetName: "rs0"

# Process Management
processManagement:
  timeZoneInfo: /usr/share/zoneinfo
```

Step 7.2: Start MongoDB

```
sudo systemctl start mongod
sudo systemctl enable mongod
sudo systemctl status mongod
```

Step 7.3: Initialize the Replica Set

⚠️ Use `mongosh`, not `mongo`! The legacy `mongo` shell is deprecated and removed in MongoDB 6.0+.

On mongodb1:

```
mongosh --host 10.10.1.122
```

Initialize:

```
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongodb1.yourdomain.com:27017", priority: 2 },
    { _id: 1, host: "mongodb2.yourdomain.com:27017", priority: 1 },
    { _id: 2, host: "mongodb3.yourdomain.com:27017", priority: 1 }
  ]
})
```

Check status:

```
rs.status()
```

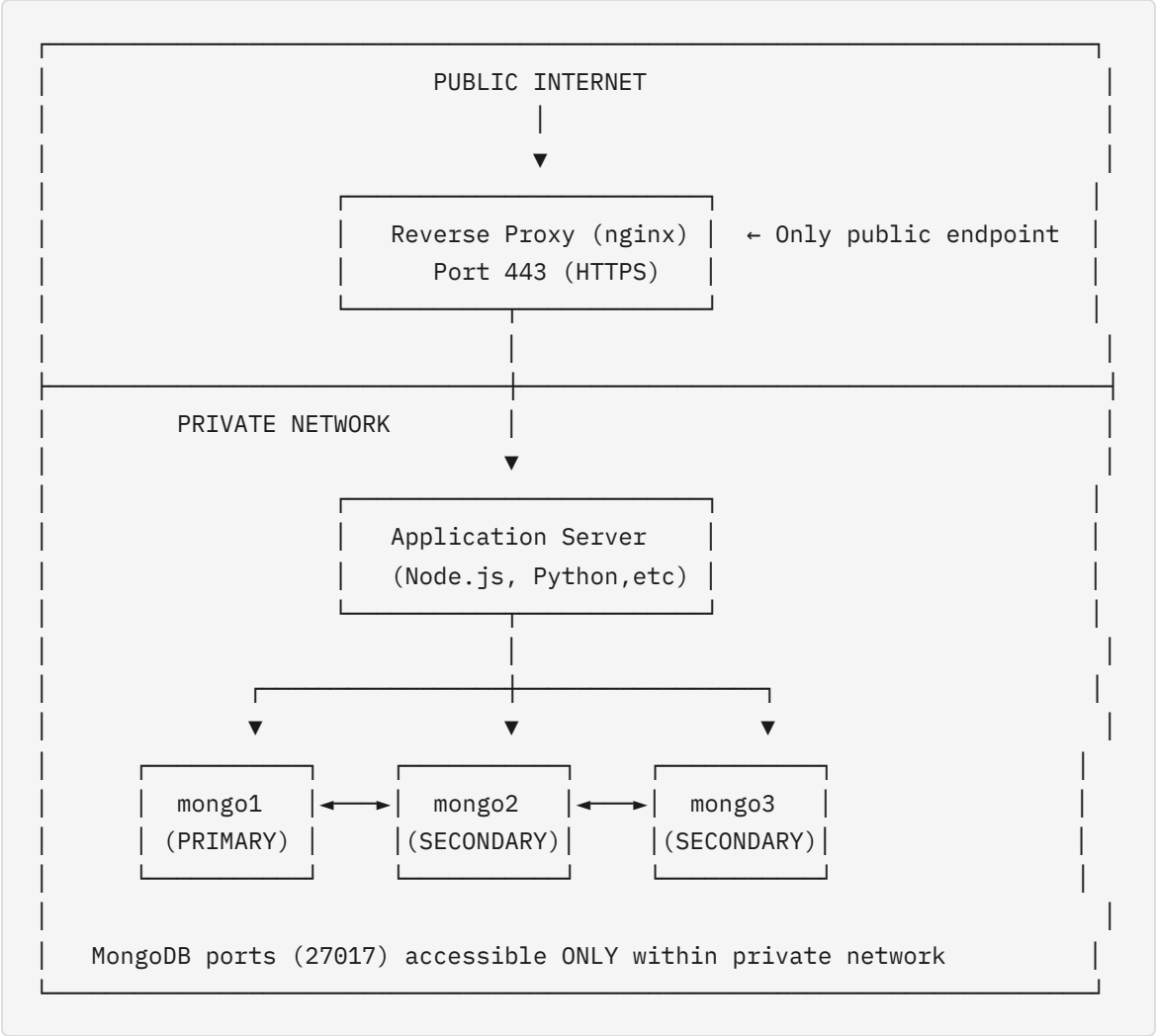
PART 8: SECURITY SETUP

Never run MongoDB in production without authentication.

🛡️ Network Architecture: Defense in Depth

Before configuring authentication, understand this critical principle: **your MongoDB server should NEVER be accessible from the public internet**. Not just "protected by authentication" – completely unreachable.

The Correct Architecture



Why This Matters

The public has **zero reason** to communicate with your MongoDB server directly. Ever. They should only interact with your application through your reverse proxy:

1. **User** → `https://yoursite.com` (nginx on port 443)
2. **Nginx** → forwards to application server (internal network)
3. **Application** → queries MongoDB (internal network)
4. Response flows back the same way

Self-Hosted: Internal DNS Configuration

For self-hosted replica sets, your MongoDB hostnames should **only resolve within your private network**:

```
# Example: Internal DNS zone (do NOT add public DNS records for these)
# These hostnames should ONLY be resolvable from within your VPC/private networ

mongodb1.internal.yourdomain.com  → 10.0.1.10  (private IP)
mongodb2.internal.yourdomain.com  → 10.0.1.11  (private IP)
mongodb3.internal.yourdomain.com  → 10.0.1.12  (private IP)

# Your replica set uses these internal hostnames:
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongodb1.internal.yourdomain.com:27017" },
    { _id: 1, host: "mongodb2.internal.yourdomain.com:27017" },
    { _id: 2, host: "mongodb3.internal.yourdomain.com:27017" }
  ]
})
```

Options for internal DNS:

- **AWS**: Use Route 53 private hosted zones
- **Docker Swarm**: Use overlay networks (automatic internal DNS)
- **Kubernetes**: Use internal service DNS
- **Self-managed**: Run your own DNS server (bind9, dnsmasq) or use `/etc/hosts`

MongoDB Atlas: IP Whitelisting

If using MongoDB Atlas, **never** whitelist `0.0.0.0/0` (allow from anywhere). Instead:

1. **Whitelist only your application server IPs**:

```
# Atlas Network Access → Add IP Address
   10.0.1.50/32    # App server 1
   10.0.1.51/32    # App server 2
```

1. **For dynamic IPs**, use Atlas Private Endpoints (AWS PrivateLink, Azure Private Link, GCP Private Service Connect)

1. **VPC Peering**: Connect your VPC directly to Atlas's VPC for fully private connectivity

Firewall Rules (Self-Hosted)

On each MongoDB server, explicitly block external access:

```
# UFW example - allow MongoDB ONLY from private network
sudo ufw default deny incoming
sudo ufw allow from 10.0.0.0/8 to any port 27017   # Private network only
sudo ufw allow from 172.16.0.0/12 to any port 27017  # Docker networks
sudo ufw deny 27017  # Deny all other MongoDB access
sudo ufw enable

# iptables example
iptables -A INPUT -p tcp --dport 27017 -s 10.0.0.0/8 -j ACCEPT
iptables -A INPUT -p tcp --dport 27017 -j DROP
```

Cloud Provider Security Groups

AWS Security Group Example:

```
Inbound Rules for MongoDB instances:
┌────────────┬────────────┬──────────────────────────────┐
│ Port       │ Protocol   │ Source                       │
├────────────┼────────────┼──────────────────────────────┤
│ 27017      │ TCP        │ sg-app-servers (not 0.0.0.0)│
│ 27017      │ TCP        │ 10.0.0.0/16 (VPC CIDR)       │
└────────────┴────────────┴──────────────────────────────┘

❌ NEVER: 27017 TCP from 0.0.0.0/0
```

Quick Checklist

- [ ] MongoDB ports (27017-27019) are NOT exposed to the internet
- [ ] MongoDB hostnames resolve only within private network
- [ ] Application servers connect to MongoDB via private IPs/hostnames
- [ ] Firewall rules explicitly deny external MongoDB access
- [ ] (Atlas) IP whitelist contains only your server IPs, not `0.0.0.0/0`
- [ ] (Atlas) Consider VPC Peering or Private Endpoints for production

---

Step 8.1: Create Admin User

On the PRIMARY:

```
use admin

db.createUser({
  user: "adminUser",
  pwd: "YourStrongPassword123!",
  roles: [{ role: "root", db: "admin" }]
})
```

Step 8.2: Generate Keyfile

```
sudo mkdir -p /keys
openssl rand -base64 756 | sudo tee /keys/mongodb.key > /dev/null
sudo chown mongodb:mongodb /keys/mongodb.key
sudo chmod 400 /keys/mongodb.key
```

Copy this keyfile to ALL nodes with the same permissions.

Step 8.3: Enable Authentication

On ALL nodes, edit `/etc/mongod.conf`:

```
security:
  authorization: enabled
  keyFile: /keys/mongodb.key
```

Restart MongoDB on all nodes:

```
sudo systemctl restart mongod
```

Step 8.4: Connect with Authentication

```
mongosh "mongodb://mongodb1.yourdomain.com:27017,mongodb2.yourdomain.com:27017,
  --username adminUser \
  --authenticationDatabase admin
```

PART 9: TLS ENCRYPTION

⚠️ **The `ssl` options are deprecated since MongoDB 4.2!** Use `tls` instead.

Step 9.1: Generate CA Certificate

```
sudo mkdir -p /keys/certs && cd /keys/certs

# Generate CA private key
sudo openssl genrsa -aes256 -out mongoCA.key 4096

# Generate CA certificate
sudo openssl req -x509 -new -extensions v3_ca -key mongoCA.key -days 365 -out m
  -subj "/C=US/ST=YourState/L=YourCity/O=YourOrg/OU=IT/CN=MongoDB-CA"
```

Step 9.2: Generate Node Certificates

For each node (example for mongodb1):

```
# Generate key and CSR
sudo openssl req -new -nodes -newkey rsa:4096 \
  -keyout mongodb1.key -out mongodb1.csr \
  -subj "/C=US/ST=YourState/L=YourCity/O=YourOrg/OU=IT/CN=mongodb1.yourdomain.c

# Sign with CA
sudo openssl x509 -CA mongoCA.crt -CAkey mongoCA.key -CAcreateserial \
  -req -days 365 -in mongodb1.csr -out mongodb1.crt

# Create PEM file
cat mongodb1.key mongodb1.crt | sudo tee mongodb1.pem > /dev/null
```

Repeat for mongodb2 and mongodb3.

Step 9.3: Enable TLS in Configuration

Update `/etc/mongod.conf`:

```
net:
  port: 27017
  bindIp: 10.10.1.122
  tls:
    mode: requireTLS
    certificateKeyFile: /keys/certs/mongodb1.pem
    CAFile: /keys/certs/mongoCA.crt
    clusterFile: /keys/certs/mongodb1.pem

security:
  authorization: enabled
  clusterAuthMode: x509
```

Step 9.4: Connect with TLS

```
mongosh --tls \
  --tlsCAFile /keys/certs/mongoCA.crt \
  --tlsCertificateKeyFile /keys/certs/mongodb1.pem \
  --host "rs0/mongodb1.yourdomain.com:27017,mongodb2.yourdomain.com:27017,mongo
  -u adminUser --authenticationDatabase admin
```

PART 10: BACKUP & RESTORE

mongodump Basics

```
# Full backup with compression
mongodump \
  --host "rs0/mongodb1.yourdomain.com:27017" \
  --username adminUser \
  --authenticationDatabase admin \
  --gzip \
  --archive=/backup/mongodb/backup_$(date +%Y%m%d).gz

# Stream directly to S3
mongodump \
  --host "rs0/mongodb1.yourdomain.com:27017" \
  --username adminUser \
  --authenticationDatabase admin \
  --gzip \
  --archive | aws s3 cp - s3://your-bucket/backup.gz
```

mongorestore Basics

```
# Restore from archive
mongorestore \
  --host "rs0/mongodb1.yourdomain.com:27017" \
  --username adminUser \
  --authenticationDatabase admin \
  --gzip \
  --archive=/backup/mongodb/backup_20250117.gz

# Drop existing before restore (DANGEROUS)
mongorestore --drop --gzip --archive=/backup/backup.gz
```

Key Limitations

- **Slow for >100GB** - Consider filesystem snapshots instead
- **Indexes must rebuild** - Restoration can be slow
- **No point-in-time without** `--oplog`

PART 11: LOG ROTATION & AUTOMATED BACKUPS

Log Rotation

```
sudo nano /etc/logrotate.d/mongod
```

Add:

```
/var/log/mongodb/*.log {
    daily
    rotate 7
    compress
    dateext
    missingok
    notifempty
    copytruncate
}
```

Automated S3 Backup Script

Create `/data/backup_to_s3.sh`:

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

REPLICA_SET="rs0"
HOSTS="mongodb1.yourdomain.com:27017,mongodb2.yourdomain.com:27017,mongodb3.you
ADMIN_USER="adminUser"
ADMIN_PASS="YourPassword"
S3_BUCKET="your-backup-bucket"
TIMESTAMP=$(date +"%Y%m%d_%H%M%S")

mongodump --host "${REPLICA_SET}/${HOSTS}" \
  --username ${ADMIN_USER} \
  --password ${ADMIN_PASS} \
  --authenticationDatabase admin \
  --tls --tlsAllowInvalidCertificates \
  --gzip \
  --archive | aws s3 cp - "s3://${S3_BUCKET}/mongodb/${TIMESTAMP}.dump.gz"
```

Schedule:

```
sudo chmod +x /data/backup_to_s3.sh
sudo crontab -e
# Add: 0 2 * * * /data/backup_to_s3.sh >> /var/log/mongodb-backup.log 2>&1
```

PART 12: THE AGGREGATION FRAMEWORK

When to Use `aggregate()` vs `find()`

USE `FIND()`	USE `AGGREGATE()`
Simple document retrieval	Complex data transformations
Single collection queries	Joining collections (`$lookup`)
Basic filtering	Grouping and summarization
When you need a cursor	Computing averages, sums, counts

Key Stages

```
// $match - Filter FIRST for performance (uses indexes!)
{ $match: { status: "active", createdAt: { $gte: ISODate("2024-01-01") } } }

// $group - Aggregate values
{ $group: {
    _id: "$category",
    total: { $sum: "$amount" },
    count: { $sum: 1 }
} }

// $lookup - Join collections
{ $lookup: {
    from: "orders",
    localField: "_id",
    foreignField: "customerId",
    as: "customerOrders"
} }

// $project - Reshape output (use LAST)
{ $project: { name: 1, total: 1, _id: 0 } }
```

Best Practices

1. **Put `$match` FIRST** - Uses indexes, reduces documents early
2. **Use `allowDiskUse: true`** for large datasets (100MB RAM limit per stage)
3. **Avoid `$unwind` on large arrays** - Creates document per element

---

PART 13: BULK WRITE OPERATIONS

Why Use Bulk Writes?

```
Individual inserts (10,000 docs): ~45 seconds
Bulk insert (10,000 docs):        ~2 seconds
```

22x improvement!

Basic Syntax

```
db.collection.bulkWrite([
  { insertOne: { document: { name: "A", price: 10 } } },
  { updateOne: {
      filter: { name: "B" },
      update: { $set: { price: 20 } }
  } },
  { deleteOne: { filter: { status: "old" } } }
], { ordered: false })  // unordered = parallel, faster
```

Best Practices

- Batch size: ~1,000 documents
- Use `ordered: false` for max throughput
- Don't exceed 100,000 operations per call

PART 14: INDEXING STRATEGIES

The ESR Rule

For compound indexes, order fields by:

1. **E**quality fields first (exact matches)
2. **S**ort fields second
3. **R**ange fields last ($gt, $lt)

```
// Query
db.orders.find({
  customerId: "abc123",      // Equality
  status: "active",          // Equality
  price: { $gte: 100 }       // Range
}).sort({ createdAt: -1 })   // Sort

// Optimal index
db.orders.createIndex({
  customerId: 1,    // E
  status: 1,        // E
  createdAt: -1,    // S
  price: 1          // R
})
```

Common Mistakes

❌ Creating an index on every field - kills write performance ❌ Ignoring index order - `{a: 1, b: 1}` ≠ `{b: 1, a: 1}` ❌ Indexing low-cardinality fields alone - `{ active: true/false }` isn't selective

---

PART 15: PROFILING & LOGGING

This is how you find slow queries and debug performance issues. But be careful - profiling itself has significant overhead that can hurt production performance.

Database Profiler

The profiler captures detailed information about database operations and writes them to the `system.profile` capped collection.

Benefits of Profiling

✅ **Identifies slow queries** - Find exactly which operations are taking too long

✅ **Shows missing indexes** - `COLLSCAN` in planSummary reveals full collection scans

✅ **Captures actual execution stats** - See `docsExamined`, `keysExamined`, execution time

✅ **Query-level granularity** - More detailed than server-wide metrics

✅ **Historical data** - Review past slow operations (while data remains in capped collection)

✅ **No application changes required** - Enable/disable without code deployment

⚠️ Disadvantages & Production Risks

Profiling has real costs. Before enabling it in production, understand the trade-offs:

RISK	IMPACT	MITIGATION
Write overhead	Every profiled op writes to `system.profile`	Use level 1 with high `slowms`, use `sampleRate`
Lock contention	Writes to `system.profile` take locks	Keep profiling sessions short
Storage I/O	Additional writes compete with your workload	Monitor disk I/O during profiling
Memory pressure	Profile docs consume WiredTiger cache	Size capped collection appropriately
Cascading slowdowns	Profiling overhead can make more queries "slow"	Start with high `slowms` threshold

Level 2 is Dangerous in Production

🚨 **NEVER use `db.setProfilingLevel(2)` in production.** Here's why:

```
Level 2 = profile EVERY operation

On a busy system doing 10,000 ops/sec:
- That's 10,000 writes/sec to system.profile
- Plus serialization overhead for each
- Plus lock acquisition for each write
- Result: You can easily DOUBLE your latency or worse
```

Real-world horror story: Enabling level 2 on a production cluster can cause a cascading failure - the profiling overhead slows queries down, which generates more slow query logs, which adds more overhead, until the system grinds to a halt.

Even Level 1 Has Overhead

Level 1 (slow queries only) is safer but not free:

```
// The profiler must evaluate EVERY operation to check if it's slow
// Even if it doesn't log, there's still:
// - Timer overhead on every op
// - Threshold comparison on every op
// - Document construction for slow ops
// - Write to capped collection for slow ops
```

Measured overhead (varies by workload):

- Level 1 with `slowms: 100`: ~1-5% overhead typical
- Level 1 with `slowms: 10`: ~5-15% overhead (more ops qualify)
- Level 2: ~50-200%+ overhead (DO NOT USE)

Production-Safe Profiling Strategy

```
// ❌ DANGEROUS: Low threshold, no sampling
db.setProfilingLevel(1, { slowms: 10 })  // Too aggressive!

// ❌ NEVER IN PRODUCTION
db.setProfilingLevel(2)  // Profiles everything!

// ✅ SAFER: High threshold
db.setProfilingLevel(1, { slowms: 100 })  // Only very slow queries

// ✅ SAFEST: High threshold + sampling (MongoDB 4.0+)
db.setProfilingLevel(1, { slowms: 50, sampleRate: 0.1 })  // 10% of slow ops

// ✅ BEST: Profile briefly, then disable
db.setProfilingLevel(1, { slowms: 100 })
// ... collect data for 5-10 minutes ...
db.setProfilingLevel(0)  // ALWAYS turn off when done!
```

Profiling Levels Reference

LEVEL	WHAT'S LOGGED	OVERHEAD	WHEN TO USE
0	Nothing	None	Production default
1	Ops slower than `slowms`	Low-Medium	Short debugging sessions
2	ALL operations	**SEVERE**	Development/testing ONLY

Enable Profiling (With Appropriate Caution)

```
// Check current status first
db.getProfilingStatus()
// Returns: { was: 0, slowms: 100, sampleRate: 1.0 }

// Enable for slow queries only (>100ms)
db.setProfilingLevel(1, { slowms: 100 })

// With sampling to reduce overhead (MongoDB 4.0+)
db.setProfilingLevel(1, { slowms: 50, sampleRate: 0.5 })  // 50% of slow ops

// DISABLE when done - don't leave it running!
db.setProfilingLevel(0)
```

Per-Database Scope

Important: Profiling level is set **per database**, not globally.

```
// This only profiles the current database
use myDatabase
db.setProfilingLevel(1, { slowms: 100 })

// Other databases are unaffected
use otherDatabase
db.getProfilingStatus()  // Still level 0

// To profile multiple databases, set each one
use db1
db.setProfilingLevel(1, { slowms: 100 })
use db2
db.setProfilingLevel(1, { slowms: 100 })
```

Query the Profiler

```
// Find slowest queries in last hour
db.system.profile.find({
  ts: { $gt: new Date(Date.now() - 3600000) }
}).sort({ millis: -1 }).limit(10)

// Find slow queries on a specific collection
db.system.profile.find({
  ns: "myDatabase.orders",
  millis: { $gt: 100 }
}).sort({ ts: -1 }).limit(10)

// Find queries that did collection scans (no index used!)
db.system.profile.find({
  planSummary: "COLLSCAN"
}).sort({ millis: -1 })

// Find queries with high document examination ratio
db.system.profile.find({
  $expr: { $gt: ["$docsExamined", { $multiply: ["$nreturned", 10] }] }
}).limit(10)
```

Understanding Profiler Output

```
{
  "op": "query",                    // Operation type
  "ns": "mydb.users",               // Namespace (database.collection)
  "command": {                       // The actual query
    "find": "users",
    "filter": { "email": "test@example.com" }
  },
  "keysExamined": 1,                // Index keys scanned
  "docsExamined": 1,                // Documents scanned
  "nreturned": 1,                   // Documents returned
  "millis": 2,                      // Execution time in ms
  "planSummary": "IXSCAN { email: 1 }",  // Index used (or COLLSCAN if none!)
  "ts": ISODate("2025-01-17...")    // Timestamp
}
```

Red flags to watch for:

- `planSummary: "COLLSCAN"` - No index used, scanning entire collection

- `docsExamined >> nreturned` - Scanning many docs to return few (need better index)
- `keysExamined >> docsExamined` - Index might not be optimal

Profiler Collection Management

```
// Check profiler collection size
db.system.profile.stats()

// The system.profile collection is a capped collection (fixed size)
// Default size varies by MongoDB version, typically 1MB

// To resize, you need to drop and recreate it:
db.setProfilingLevel(0)
db.system.profile.drop()
db.createCollection("system.profile", { capped: true, size: 10485760 })  // 10M
db.setProfilingLevel(1, { slowms: 100 })
```

Log Verbosity with setLogLevel

For more granular control over what MongoDB logs, use `db.setLogLevel()`. This is useful when you need detailed logging for specific components without enabling full profiling.

Log Components

```
// Set verbosity for specific component (0-5, higher = more verbose)
db.setLogLevel(1, "command")      // Log all commands
db.setLogLevel(1, "query")        // Log query planning
db.setLogLevel(1, "write")        // Log write operations
db.setLogLevel(1, "index")        // Log index operations
db.setLogLevel(1, "replication")  // Log replication activity
db.setLogLevel(1, "network")      // Log network activity
db.setLogLevel(1, "storage")      // Log storage operations

// Reset to default
db.setLogLevel(0, "command")

// Check current log levels
db.adminCommand({ getParameter: 1, logComponentVerbosity: 1 })
```

Log Levels Explained

LEVEL	DESCRIPTION
0	Default - Informational messages only
1	Debug - Includes debug messages
2	Debug - More verbose
3-5	Increasingly verbose (rarely needed)

Practical Examples

```
// Debugging slow commands in production (temporary)
db.setLogLevel(1, "command")
// ... reproduce the issue ...
db.setLogLevel(0, "command")  // Don't forget to turn off!

// Debugging replication lag
db.setLogLevel(2, "replication")

// Debugging index build issues
db.setLogLevel(1, "index")

// View logs (in another terminal)
// sudo tail -f /var/log/mongodb/mongod.log | grep COMMAND
```

Setting Log Levels in Configuration

You can also set log verbosity in `mongod.conf`:

```
systemLog:
  destination: file
  path: /var/log/mongodb/mongod.log
  logAppend: true
  verbosity: 0  # Global default
  component:
    command:
      verbosity: 1    # Log commands
    query:
      verbosity: 0
    replication:
      verbosity: 1    # Log replication
```

Combining Profiler and Logs

Recommended production debugging workflow:

1. **Start with profiler** for slow query identification:

```
db.setProfilingLevel(1, { slowms: 100 })
```

1. **Query system.profile** to find problematic queries:

```
db.system.profile.find({ millis: { $gt: 100 } }).sort({ millis: -1 }).limit(5)
```

1. **Use explain()** on identified slow queries:

```
db.orders.find({ customerId: "abc" }).explain("executionStats")
```

1. **Enable command logging temporarily** if you need more context:

```
db.setLogLevel(1, "command")
   // Reproduce issue
   db.setLogLevel(0, "command")
```

1. **Disable profiler when done:**

```
db.setProfilingLevel(0)
```

Alternatives to Profiling (Lower Overhead Options)

Before reaching for the profiler, consider these lower-impact alternatives:

1. MongoDB Log Slow Queries (Zero Overhead When No Slow Queries)

MongoDB already logs slow operations by default! Check `mongod.conf`:

```
operationProfiling:
  slowOpThresholdMs: 100  # Log ops slower than 100ms
  mode: off               # Don't use profiler, just log
```

Then grep the logs:

```
# Find slow queries in logs
sudo grep "Slow query" /var/log/mongodb/mongod.log

# Or with more context
sudo grep -E "(COMMAND|QUERY).*[0-9]{3,}ms" /var/log/mongodb/mongod.log
```

Advantage: No writes to `system.profile`, lower overhead than profiling.

2. currentOp() for Real-Time Slow Queries

```
// Find currently running slow operations (no profiler needed)
db.currentOp({
  "secs_running": { $gt: 5 },    // Running longer than 5 seconds
  "op": { $ne: "none" }
})

// Find operations waiting on locks
db.currentOp({ "waitingForLock": true })
```

Advantage: Shows what's happening RIGHT NOW, zero storage overhead.

3. $indexStats for Unused Index Detection

```
// Find indexes that aren't being used (no profiler needed)
db.orders.aggregate([{ $indexStats: {} }])

// Look for indexes with low 'accesses.ops' count
```

Advantage: Helps optimize indexes without profiling overhead.

4. explain() for Specific Query Analysis

```
// Analyze a specific query's execution plan
db.orders.find({ customerId: "abc" }).explain("executionStats")

// Key metrics to check:
// - totalDocsExamined vs nReturned
// - executionTimeMillis
// - stage (COLLSCAN = bad, IXSCAN = good)
```

Advantage: Targeted analysis without profiling all operations.

5. Atlas/Cloud Manager Performance Advisor

If you're on Atlas or using Cloud Manager, use the **Performance Advisor** instead of profiling - it analyzes query patterns and suggests indexes with minimal overhead.

When You DO Need the Profiler

Use `db.setProfilingLevel()` when:

- Log-based analysis isn't detailed enough
- You need to capture the exact query shape
- You're debugging intermittent issues
- You need the actual `docsExamined` / `keysExamined` stats

Even then, **keep sessions short** and **always disable when done**.

Quick Reference: Finding Performance Issues

```
// === STEP 1: Enable profiler ===
db.setProfilingLevel(1, { slowms: 100 })

// === STEP 2: Find slow queries ===
// Top 10 slowest queries
db.system.profile.find().sort({ millis: -1 }).limit(10).pretty()

// Queries without indexes (COLLSCAN)
db.system.profile.find({ planSummary: /COLLSCAN/ }).pretty()

// Queries examining way more docs than returned
db.system.profile.aggregate([
  { $match: { docsExamined: { $gt: 0 } } },
  { $project: {
      ns: 1,
      millis: 1,
      ratio: { $divide: ["$docsExamined", { $add: ["$nreturned", 1] }] }
  }},
  { $match: { ratio: { $gt: 100 } } },  // 100:1 ratio = bad
  { $sort: { ratio: -1 } },
  { $limit: 10 }
])

// === STEP 3: Analyze specific query ===
db.orders.find({ customerId: "abc" }).explain("executionStats")
// Look for: totalDocsExamined vs nReturned

// === STEP 4: Check index usage ===
db.orders.aggregate([{ $indexStats: {} }])

// === STEP 5: Disable when done ===
db.setProfilingLevel(0)
```

PART 16: ACID TRANSACTIONS

When Do You Need Transactions?

MongoDB provides **single-document atomicity** by default. Use multi-document transactions only when you need atomic updates across **multiple documents or collections**.

```
const session = client.startSession();

try {
  session.startTransaction();

  await accounts.updateOne(
    { _id: fromAccountId },
    { $inc: { balance: -amount } },
    { session }
  );

  await accounts.updateOne(
    { _id: toAccountId },
    { $inc: { balance: amount } },
    { session }
  );

  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
  throw error;
} finally {
  session.endSession();
}
```

Best Practices

- Keep transactions under 60 seconds
- Limit to ~1,000 document modifications
- Use proper indexes (slow queries hold locks longer)
- Most operations (80-90%) don't need transactions

PART 17: AWS/CLOUD HOSTING COSTS

Self-Hosted vs Atlas (2025)

WORKLOAD	ATLAS	SELF-HOSTED (ON-DEMAND)	SELF-HOSTED (RESERVED)
8GB RAM (M30)	~$390/mo	~$200/mo	~$130/mo
32GB RAM (M50)	~$1,440/mo	~$530/mo	~$350/mo
64GB RAM (M60)	~$2,847/mo	~$900/mo	~$600/mo

Hidden self-hosting costs: Engineering time (~$400-800/mo), monitoring, on-call burden.

Break-even: ~$2,000-3,000/month Atlas spend.

PART 18: TROUBLESHOOTING

Common Issues

"not reachable/healthy" errors:

- Check firewall rules (port 27017)
- Verify hostnames resolve on all nodes
- Check `bindIp` in mongod.conf

Authentication failures:

- Verify keyfile is identical on all nodes
- Check keyfile permissions (must be 400)
- Ensure keyfile owner is `mongodb`

Replica set won't initialize:

- Ensure all nodes have the same `replSetName`
- Use FQDNs, not IP addresses (MongoDB 5.0+)

Useful Commands

```
# MongoDB logs
sudo tail -f /var/log/mongodb/mongod.log

# Filter for replication issues
sudo tail /var/log/mongodb/mongod.log | grep repl
```

```
// Replica set status
rs.status()
rs.conf()
rs.printSecondaryReplicationInfo()

// Performance
db.currentOp()
db.serverStatus()
```

PART 19: MONITORING & ALERTING

You can't fix what you can't see. Production MongoDB requires proper monitoring.

Key Metrics to Monitor

METRIC	WARNING THRESHOLD	CRITICAL THRESHOLD	WHY IT MATTERS
Replication Lag	> 10 seconds	> 60 seconds	Secondaries falling behind = data loss risk
Connections	> 80% of max	> 90% of max	Connection exhaustion = app failures
Query Targeting	ratio > 100	ratio > 1000	High doc scan to return ratio = missing indexes
Page Faults	Increasing trend	Sustained high	Working set exceeds RAM
Oplog Window	< 24 hours	< 1 hour	Too small = replication/recovery problems
Disk Usage	> 70%	> 85%	Running out of space = writes fail
CPU Usage	> 70% sustained	> 90%	May need to scale or optimize
Cache Usage	> 80%	> 95%	WiredTiger cache pressure

Essential Commands for Monitoring

```
// Replication lag (run on primary)
rs.printSecondaryReplicationInfo()

// Connection count
db.serverStatus().connections
// { current: 45, available: 51155, totalCreated: 1234 }

// WiredTiger cache stats
db.serverStatus().wiredTiger.cache
// Look for: "bytes currently in the cache"
// Compare to: "maximum bytes configured"

// Oplog size and window
db.getReplicationInfo()
// { logSizeMB: 990, usedMB: 456, timeDiff: 172800, ... }
// timeDiff = oplog window in seconds (172800 = 48 hours)

// Current operations (find long-running queries)
db.currentOp({ "secs_running": { $gt: 5 } })

// Index stats (find unused indexes)
db.collection.aggregate([{ $indexStats: {} }])
```

Prometheus + Grafana Setup (Recommended)

Use the **MongoDB Exporter** for Prometheus:

```
# Install mongodb_exporter
wget https://github.com/percona/mongodb_exporter/releases/download/v0.40.0/mong
tar xvzf mongodb_exporter-0.40.0.linux-amd64.tar.gz

# Run exporter
./mongodb_exporter --mongodb.uri="mongodb://monitor:password@localhost:27017/ad
```

Critical Grafana Dashboards:

- Percona MongoDB Dashboard (ID: 2583)
- MongoDB Overview (ID: 12079)

Alerting Rules (Examples)

```
# Prometheus alerting rules
groups:
  - name: mongodb
    rules:
      - alert: MongoDBReplicationLag
        expr: mongodb_rs_members_optimeDate{state="SECONDARY"} - mongodb_rs_mem
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "MongoDB replication lag > 60 seconds"

      - alert: MongoDBConnectionsHigh
        expr: mongodb_ss_connections{conn_type="current"} / mongodb_ss_connecti
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "MongoDB connections > 80% of max"

      - alert: MongoDBDown
        expr: up{job="mongodb"} == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "MongoDB instance is down"
```

Quick Health Check Script

```
#!/bin/bash
# mongodb-health-check.sh

mongosh --quiet --eval '
  const status = rs.status();
  const primary = status.members.find(m => m.stateStr === "PRIMARY");
  const secondaries = status.members.filter(m => m.stateStr === "SECONDARY");

  print("=== Replica Set Health ===");
  print("Primary: " + (primary ? primary.name : "NONE!"));
  print("Secondaries: " + secondaries.length);

  // Check replication lag
  secondaries.forEach(s => {
    const lag = (primary.optimeDate - s.optimeDate) / 1000;
    print("  " + s.name + " lag: " + lag + "s" + (lag > 60 ? " ⚠️ HIGH!" : " ✓"
  });

  // Check connections
  const conn = db.serverStatus().connections;
  const connPct = (conn.current / (conn.current + conn.available) * 100).toFixe
  print("Connections: " + conn.current + " (" + connPct + "%)");

  // Check oplog window
  const oplog = db.getReplicationInfo();
  const oplogHours = (oplog.timeDiff / 3600).toFixed(1);
  print("Oplog window: " + oplogHours + " hours" + (oplogHours < 24 ? " ⚠️ LOW!
'
```

PART 20: CONNECTION POOLING & READ/WRITE CONCERNS

Connection Pooling

Connections are expensive. Each connection consumes ~1MB of RAM on the server. Poor connection management is a common cause of MongoDB performance issues.

Connection String Best Practices

```
// ❌ BAD: No pool settings, defaults may not fit your workload
const uri = "mongodb://mongodb1:27017,mongodb2:27017,mongodb3:27017/?replicaSet

// ✅ GOOD: Explicit pool configuration
const uri = "mongodb://mongodb1:27017,mongodb2:27017,mongodb3:27017/?" +
  "replicaSet=rs0" +
  "&maxPoolSize=100" +        // Max connections per server
  "&minPoolSize=10" +         // Keep minimum connections warm
  "&maxIdleTimeMS=60000" +    // Close idle connections after 60s
  "&waitQueueTimeoutMS=5000" + // Fail fast if pool exhausted
  "&retryWrites=true" +       // Auto-retry transient write failures
  "&retryReads=true" +        // Auto-retry transient read failures
  "&w=majority" +             // Default write concern
  "&readPreference=secondaryPreferred"  // Read from secondaries when possible
```

Pool Sizing Guidelines

APPLICATION TYPE	RECOMMENDED `MAXPOOLSIZE`	NOTES
Web app (low traffic)	10-50	Default is usually fine
Web app (high traffic)	50-100	Per-server, not total
Background workers	5-20	Usually need fewer connections
Microservices	10-30	Each service has own pool

Formula: `maxPoolSize` × number of app servers < `net.maxIncomingConnections` (default 65536)

Diagnosing Connection Issues

```
// Check current connections
db.serverStatus().connections
// { current: 245, available: 51000, totalCreated: 89234 }

// Find connections by client
db.currentOp(true).inprog.reduce((acc, op) => {
  const client = op.client || "unknown";
  acc[client] = (acc[client] || 0) + 1;
  return acc;
}, {})
```

Read Preference

Controls which replica set members handle read operations.

MODE	READS FROM	USE CASE
`primary`	Primary only	Default. Strongest consistency
`primaryPreferred`	Primary, fallback to secondary	Consistency with HA
`secondary`	Secondaries only	Offload reads from primary
`secondaryPreferred`	Secondaries, fallback to primary	Best for read scaling
`nearest`	Lowest latency member	Geo-distributed apps

```
// In connection string
"mongodb://.../?readPreference=secondaryPreferred"

// Per-query
db.orders.find({}).readPref("secondary")

// With tags (read from specific datacenter)
db.orders.find({}).readPref("secondary", [{ dc: "east" }])
```

⚠️ **Warning:** Reading from secondaries means potentially stale data (replication lag).

Write Concern

Controls acknowledgment level for write operations.

WRITE CONCERN	MEANING	DURABILITY	PERFORMANCE
`w: 0`	No acknowledgment	❌ None	Fastest
`w: 1`	Primary acknowledged	⚠️ Weak	Fast
`w: "majority"`	Majority acknowledged	✅ Strong	Slower
`w: 3`	3 members acknowledged	✅ Strong	Slowest

```
// In connection string (default for all writes)
"mongodb://.../?w=majority&wtimeoutMS=5000"

// Per-operation
db.orders.insertOne(
  { item: "widget" },
  { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

Recommended Settings

USE CASE	WRITE CONCERN	READ PREFERENCE
Financial/critical data	`w: "majority"`	`primary`
General application	`w: "majority"`	`primaryPreferred`
Analytics/reporting	`w: 1`	`secondaryPreferred`
Logging/metrics	`w: 1`	`secondary`

PART 21: SCHEMA DESIGN BEST PRACTICES

MongoDB's flexible schema is powerful but dangerous. Poor schema design is the #1 cause of MongoDB performance problems.

Embedding vs. Referencing

APPROACH	WHEN TO USE	EXAMPLE
Embed	Data accessed together, 1:1 or 1:few relationships	Order with line items
Reference	Large/unbounded arrays, many:many relationships	User's blog posts

```
// ✅ EMBED: Order with items (accessed together, bounded)
{
  _id: ObjectId("..."),
  customerId: "cust123",
  items: [
    { sku: "WIDGET-1", qty: 2, price: 10.00 },
    { sku: "GADGET-2", qty: 1, price: 25.00 }
  ],
  total: 45.00
}

// ✅ REFERENCE: User with posts (posts are large, unbounded)
// users collection
{ _id: "user123", name: "Alice", email: "alice@example.com" }

// posts collection
{ _id: ObjectId("..."), userId: "user123", title: "...", body: "..." }
```

Document Size Limits

LIMIT	VALUE	WHAT HAPPENS
Max document size	**16MB**	Write fails with error
Recommended max	**< 1MB**	Larger = slower operations
Array elements	No hard limit	But affects performance

Rule of thumb: If an array can grow unbounded, don't embed it.

```
// ❌ BAD: Unbounded array (will eventually hit 16MB or cause slowdowns)
{
  _id: "user123",
  activityLog: [
    { ts: ISODate("..."), action: "login" },
    { ts: ISODate("..."), action: "view_page" },
    // ... millions of entries over time
  ]
}

// ✅ GOOD: Separate collection for unbounded data
// users collection
{ _id: "user123", name: "Alice" }

// activity collection (with TTL index for auto-cleanup)
{ userId: "user123", ts: ISODate("..."), action: "login" }
```

Anti-Patterns to Avoid

❌ **Massive arrays** - Arrays that grow unbounded

❌ **Deeply nested documents** - Hard to query and index

❌ **Storing files in documents** - Use GridFS or S3

❌ **Over-normalization** - Too many collections requiring joins

❌ **No indexes on query fields** - Always index fields you filter/sort by

❌ **$where and JavaScript execution** - Extremely slow, can't use indexes

Schema Validation

Enforce structure with JSON Schema validation:

```
db.createCollection("orders", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["customerId", "items", "total"],
      properties: {
        customerId: {
          bsonType: "string",
          description: "Required customer ID"
        },
        items: {
          bsonType: "array",
          minItems: 1,
          items: {
            bsonType: "object",
            required: ["sku", "qty", "price"],
            properties: {
              sku: { bsonType: "string" },
              qty: { bsonType: "int", minimum: 1 },
              price: { bsonType: "decimal" }
            }
          }
        },
        total: { bsonType: "decimal" }
      }
    }
  },
  validationLevel: "moderate",  // Only validate inserts and updates
  validationAction: "error"     // Reject invalid documents
})
```

PART 22: DISASTER RECOVERY & FAILOVER

Testing Failover (Do This Regularly!)

Never wait for a real outage to discover your failover doesn't work.

```
// On the PRIMARY - force a stepdown (triggers election)
rs.stepDown(60)  // Steps down for 60 seconds

// Watch the election happen
rs.status()
```

Failover Test Checklist

- [ ] Application reconnects automatically
- [ ] Write operations resume after brief pause
- [ ] No data loss (check write concern = majority)
- [ ] Alerts fire correctly
- [ ] Recovery time is acceptable (< 30 seconds typical)

Oplog Sizing

The oplog determines how long you have to recover a secondary or do point-in-time recovery.

```
// Check current oplog size and window
db.getReplicationInfo()
// {
//   logSizeMB: 990,
//   usedMB: 456,
//   timeDiff: 172800,  // 48 hours of operations
//   tFirst: "...",
//   tLast: "..."
// }
```

Sizing guidelines:

- Minimum: 24 hours of operations
- Recommended: 48-72 hours
- High-write workloads: May need more

```
// Resize oplog (MongoDB 4.0+)
db.adminCommand({ replSetResizeOplog: 1, size: 16000 })  // 16GB
```

Rolling Upgrades (Zero Downtime)

Order matters! Always upgrade in this sequence:

1. **Upgrade secondaries first** (one at a time)
2. **Step down primary** → elect new primary
3. **Upgrade the old primary** (now a secondary)

4. **Optional:** Step up preferred primary

```
# On each SECONDARY (one at a time):
sudo systemctl stop mongod
# Update packages
sudo apt-get update && sudo apt-get install -y mongodb-org
sudo systemctl start mongod
# Wait for it to sync before moving to next node
mongosh --eval 'rs.status()'

# After all secondaries upgraded:
# On PRIMARY:
mongosh --eval 'rs.stepDown()'
# Wait for election, then upgrade this node same as secondaries
```

Point-in-Time Recovery

If you backed up with `--oplog`, you can restore to any point:

```
# Restore to specific timestamp
mongorestore \
  --oplogReplay \
  --oplogLimit "1705500000:1"  # Unix timestamp:increment
  --gzip --archive=/backup/backup.gz
```

PART 23: MONGODB MANAGEMENT TOOLS

Quick Comparison

TOOL	PRICE	BEST FOR
MongoDB Compass	Free	Beginners, official support
Studio 3T	$149-699/yr	Power users, SQL support
Robo 3T	Free	Quick queries, shell users
NoSQLBooster	Free-$129	Developers, SQL familiarity

QUICK REFERENCE CARD

Key Commands

```
# Service management
sudo systemctl start|stop|restart|status mongod

# Connection
mongosh "mongodb://mongodb1:27017,mongodb2:27017,mongodb3:27017/?replicaSet=rs0
```

Key Differences from Old Guides

OLD (PRE-2024)	NEW (2025)
`apt-key add`	`gpg --dearmor` to keyring
`mongo` shell	`mongosh` shell
`ssl` options	`tls` options
IP addresses	DNS hostnames required
Ubuntu 18.04	Ubuntu 22.04/24.04

CONCLUSION

You now have a production-ready MongoDB 8.0 replica set with:

✅ Modern Ubuntu (22.04/24.04) with proper package management
✅ XFS filesystem optimized for WiredTiger
✅ OS-level tuning (THP, swappiness, limits)
✅ Authentication and keyfile security
✅ TLS encryption (not deprecated SSL)
✅ Automated backups to S3
✅ Profiling and logging for performance debugging (with production safety

guidelines)
✅ Understanding of aggregation, bulk writes, indexes, and transactions
✅ Monitoring and alerting setup
✅ Connection pooling and read/write concern best practices
✅ Schema design guidelines
✅ Disaster recovery and failover procedures

Your operational checklist:

- [ ] Set up monitoring (Prometheus + MongoDB Exporter + Grafana)
- [ ] Configure alerting for replication lag, connections, disk usage
- [ ] Test failover scenarios with `rs.stepDown()` - do this monthly!
- [ ] Verify backup restoration works - test quarterly at minimum
- [ ] Document your runbooks (startup, shutdown, failover, restore)
- [ ] Review slow query logs weekly
- [ ] Audit index usage monthly (`$indexStats`)

Common post-deployment mistakes to avoid:

- ❌ Leaving profiler enabled (turn it off when done!)
- ❌ Never testing failover until a real outage
- ❌ Ignoring replication lag alerts
- ❌ Using `w: 1` for critical data (use `w: "majority"`)
- ❌ Not monitoring oplog window size
- ❌ Forgetting to test backup restores

Questions? Drop them in the comments! Happy to help debug.

Edit: Updated for MongoDB 8.0 and modern Ubuntu. Added comprehensive sections on profiling risks, monitoring, connection pooling, schema design, and disaster recovery. Thanks for the feedback!