READ MODE · ANSWER FIRST

OpenSearch 100 GB system design interview guide

Design a 100 GB OpenSearch system with SLOs, mappings, capacity, bulk backpressure, shard placement, query execution, recovery, and tradeoffs.

CHEAT SHEET · 9 ESSENTIAL IDEAS

The whole story in 9 lines

State SLOs, model index cost, bound ingestion, separate shard copies, coordinate searches, and reserve capacity for maintenance and...

1. Workload, latency, freshness, growth, and failure goals define the system before machine count does.
2. Explicit mappings and compatible analyzers preserve exact filters, full text matching, scoring, and phrases.
3. Posting lists eliminate unrelated documents early while retaining frequency and position evidence.
4. Capacity includes measured index expansion, replica copies, free disk, bounded bulks, and retry headroom.
5. Replicas protect a shard only when copies occupy independent failure domains with surviving capacity.
6. Query fan-out returns local top scores, the coordinator merges them, and fetch retrieves only winners.
7. Refresh controls visibility while merge pressure controls sustainable indexing and query cost.
8. Promotion restores ownership, then throttled recovery and rebalancing restore redundancy without crushing service.
9. Every tuning lever spends another resource, so mixed-load benchmarks and production signals decide the tradeoff.

STAGE 1 OF 11

Setup

Put the whole collection on one drive and ask for blue axolotl. A full scan can eventually find it, but our service also owes 250 searches each second, fresh writes and complete results after a zone loss. That gap is the mystery.

The first obstacle is language itself. A document may contain “Blue,” while a query sends “BLUE.” An analyzer can turn both into the same stable term, giving the system a key—but a key still needs somewhere useful to lead.

A posting supplies that destination by linking one term to a matching document ID, along with evidence such as frequency and position. This reverse direction hints at an escape from scanning, although we have not yet proved how much work it saves.

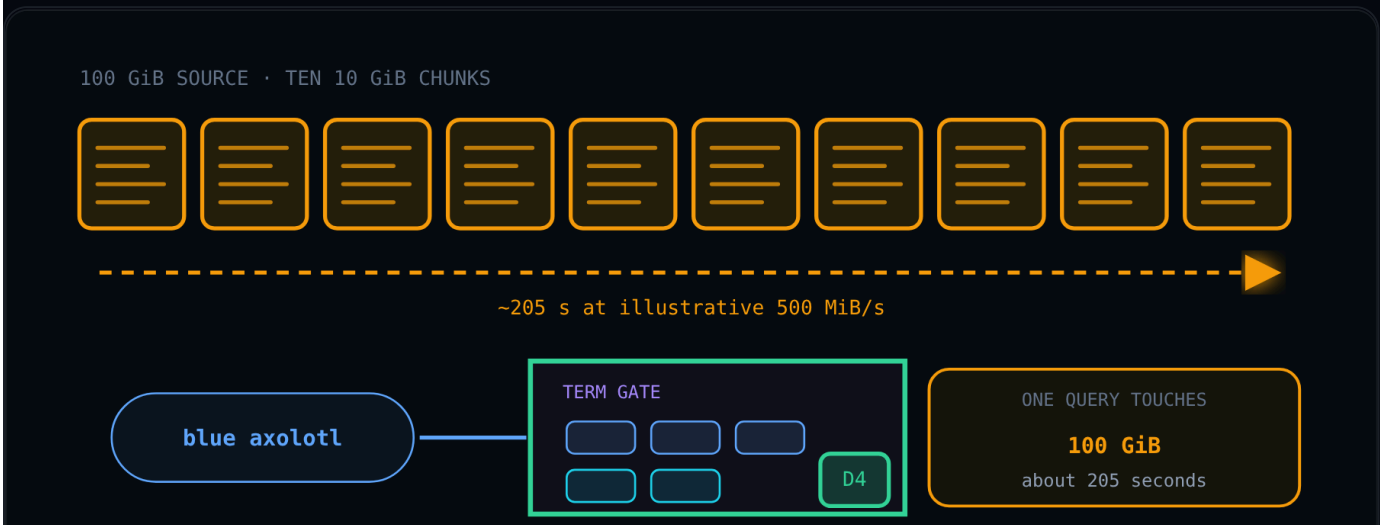
If the reverse map becomes too large or busy for one place, OpenSearch can split the documents among shards. Each shard is a complete Lucene index for its slice, which buys distribution but leaves no shard with the whole collection.

Continuous writes create a different problem: a shard cannot rebuild one enormous index file for every document. Lucene instead publishes immutable segment files, so readers keep a stable view while new searchable pieces arrive.

These four ideas are clues, not a finished architecture. To learn which ones we really need—and how many copies and machines they require—we first have to make the workload precise enough to break the single-disk design.

Frame requirements and workload

Try it in Watch mode. Choose the method that avoids rereading all 100 GB for every query.



02 FRAME REQUIREMENTS AND WORKLOAD

To test the single-disk picture, this frame divides the 100 GiB source into ten equal blocks. The blocks show exactly how much text we own, but not how often it is searched, how fresh it must be, or what failure it must survive.

With only those source blocks, the machine has no route to a likely match. At the illustrative 500 MiB per second shown here, one sequential pass through all 100 GiB takes about 204.8 seconds before competing searches add any delay.

Our request contains only blue and axolotl, yet the scan still reads every block because its work follows collection size rather than the requested terms. Which data structure can reverse that relationship?

PAUSE AND PREDICT

Which structure makes lookup work follow query terms?

- A second source copy
- An inverted index
- Compression alone

Skip this check

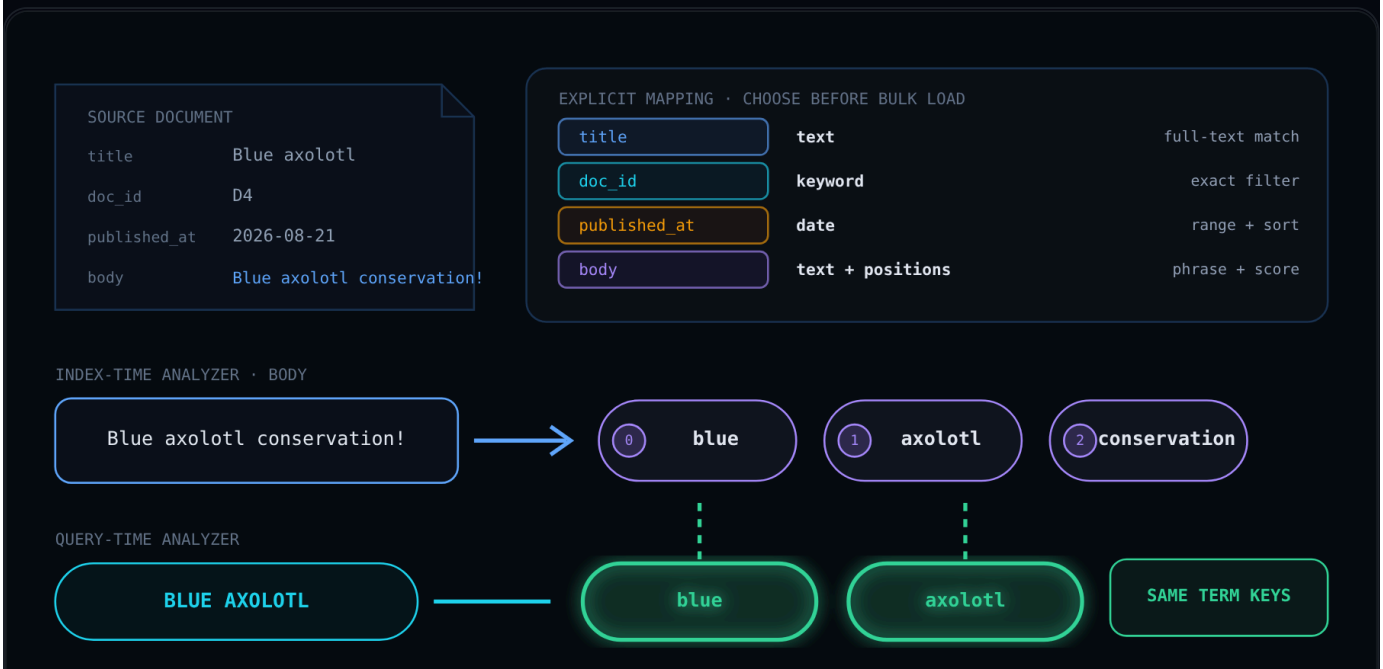
An inverted index pays the organizing cost when documents are written. The two requested terms open two ordered posting lists, and their shared ID points to candidate D4 without sending the source blocks through another scan.

Switch the Method control between Full scan and Inverted index. Full scan touches 100 GiB; Inverted index follows two posting lists to D4 while the latency, freshness and zone-loss targets stay fixed.

We have escaped work that grows with every source byte, but D4 appears only if indexing and search agree on what blue and axolotl mean. The shortcut therefore turns field meaning from a detail into the next correctness problem.

STAGE 3 OF 11

Choose mappings and analyzers



03 CHOOSE MAPPINGS AND ANALYZERS

Candidate D4 is a JSON document, not one undifferentiated sentence. Its title and body need language matching, doc_id must remain exactly D4, and published_at must still behave like a date for filtering and sorting.

The mapping records those different jobs before indexing starts. Text fields are analyzed, keyword fields preserve exact values, date fields keep time semantics, and text positions may be retained when word order matters.

For D4’s body, analysis turns “Blue axolotl conservation!” into the lowercase terms blue, axolotl and conservation at positions zero, one and two. The sentence changes form, but its searchable words and order survive.

The query takes a separate trip through an analyzer before dictionary lookup. If it produces different keys from the blue and axolotl entries stored for D4, what becomes of a document that should have matched?

PAUSE AND PREDICT

If index and query analysis disagree, what happens?

- Matches can disappear
- Ranking gets faster
- A replica repairs terms

Skip this check

Here both paths produce blue and axolotl, so the query can address the exact term entries created for D4. Its ID and date keep their own exact behavior because those fields never needed to become language tokens.

That agreement is built into the stored index and may require reindexing to repair, so representative documents should test it before a bulk load. With the keys settled, we can finally examine what OpenSearch stores under them.

Invert terms into postings

★ If you remember one thing · Two compact posting lists reveal D4 without scanning the four source documents.

04 INVERT TERMS INTO POSTINGS

To see what those agreed keys buy us, this frame condenses the fixture into four document fingerprints. The two circles on each card mark whether that document contains blue, axolotl, both, or neither.

Turning that document-first view around creates a row for blue. It points to D1, D2 and D4; the row length gives document frequency, while each posting can retain frequency and position evidence for its document.

The axolotl row is shorter, reaching only D3 and D4. Every ID absent from this row is a document that later phrase checks and scoring can ignore before touching its stored body.

Because both posting lists are ordered, the search can advance through them until their IDs agree rather than compare every possible pair. Which document survives that meeting?

PAUSE AND PREDICT

Which document appears in both posting lists?

- D1
- D3
- D4

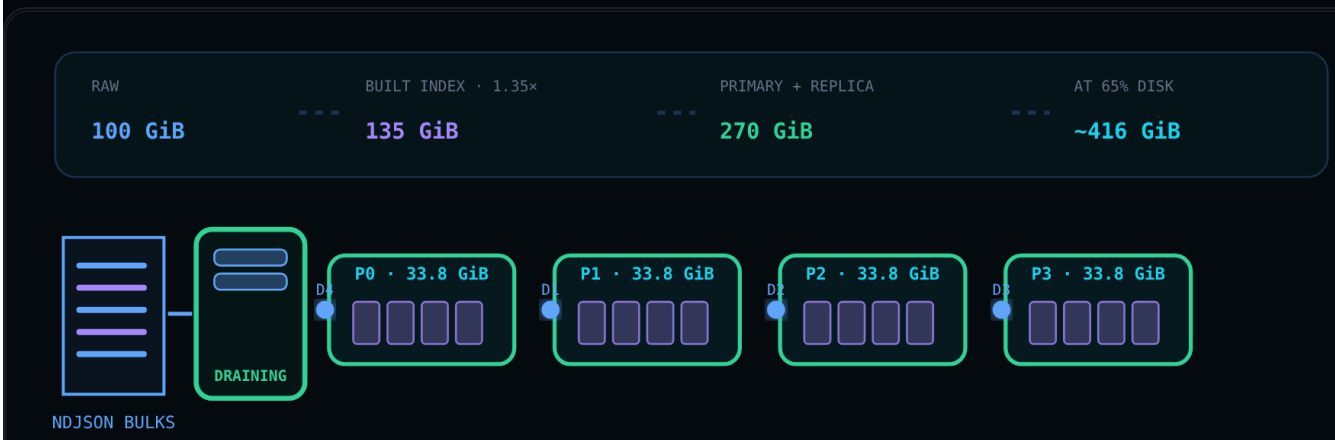
```
Skip this check
```

The rows meet at D4. D1 and D2 drop out because they lack axolotl, while D3 lacks blue; none of the original source blocks passes through this intersection.

D4 is only the membership answer. Frequencies and positions still support BM25 scoring and phrase order, and the reverse map holding all of that evidence is a real data structure that must be built, copied and kept current.

STAGE 5 OF 11

Size capacity and bound ingestion



05 SIZE CAPACITY AND BOUND INGESTION

That search shortcut is not free space beside the original text. With the illustrative measured expansion of 1.35, 100 GiB becomes 135 GiB of primary index data; one replica makes 270 GiB, and keeping disk at 65 percent raises the base provision to about 416 GiB.

The worked design divides the 135 GiB among four primary shards, about 33.8 GiB per Lucene index. Smaller shards can add parallelism and shorten recovery units, but every additional shard also adds files, memory, queues and coordination.

Documents reach those primaries in newline-delimited bulk requests, sharing request overhead without becoming one all-or-nothing result. OpenSearch reports each item separately, so successful writes can stay finished when another item fails.

During a surge, the bounded write queue fills and rejects more work instead of quietly consuming memory forever. What should the client do so that this protective refusal does not become an even larger storm?

PAUSE AND PREDICT

What should a bulk client do after overload rejection?

- Retry with backoff
- Double every batch
- Ignore item failures

Skip this check

Successful items remain complete, while rejected items wait and return selectively with backoff. The queue is not failed storage; it is a boundary protecting the memory and disk bandwidth that live searches still need.

The lower row reveals what doubled the storage estimate: each primary shard has one complete replica in another zone. Even this 416 GiB base still leaves expected growth, snapshots and measured workload headroom outside the calculation.

We have paid for two copies of every shard, but copy count alone does not say whether one failure can erase both. The protection becomes real only when those expensive bytes occupy independent failure domains.

STAGE 6 OF 11

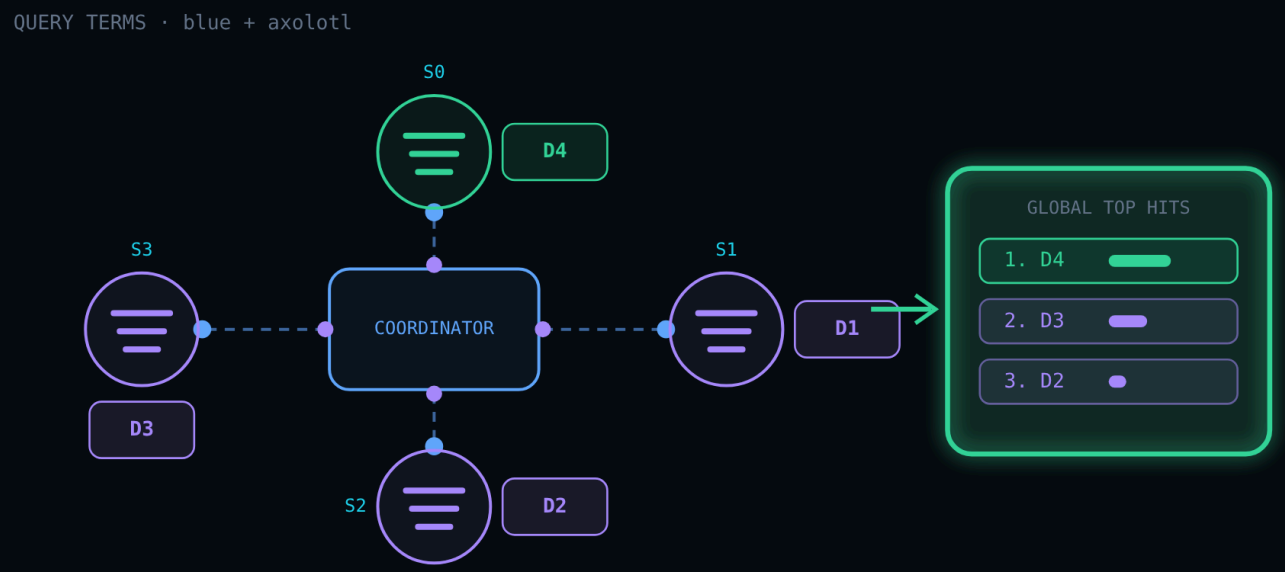
Place shards across failure domains

06 PLACE SHARDS ACROSS FAILURE DOMAINS

Shard 0 now has a primary copy named P0 and a replica named R0. Move R0 beside P0 in Zone A and fail the zone, then place R0 across the boundary and repeat. The count stays at two, yet complete service survives only when the copies do not share one failure. Those surviving shards still know only their own documents, so ranking the whole index becomes our next problem.

STAGE 7 OF 11

Search, score, coordinate, and fetch



07 SEARCH, SCORE, COORDINATE, AND FETCH

Splitting the index created four separate Lucene views, so none of them can rank the whole collection. This new picture begins at the coordinating node, where blue axolotl arrives before the evidence has been gathered from those shards.

The coordinator fans the query out to one eligible copy of each shard. When both primary and replica are available, adaptive replica selection can favor the copy with better recent response time, coordinator latency and search-queue depth.

Each selected shard opens its own posting lists and scores local candidates. BM25 uses evidence including term frequency, document frequency and document length, so a shard can return competitive IDs and scores without sending every matching body.

Four compact local rankings return to the coordinator, but they were produced independently and still need one global order. Once those scores share a list, which document should lead?

PAUSE AND PREDICT

Which result should rank first for blue axolotl?

- D1, blue only
- D3, axolotl only
- D4, both terms

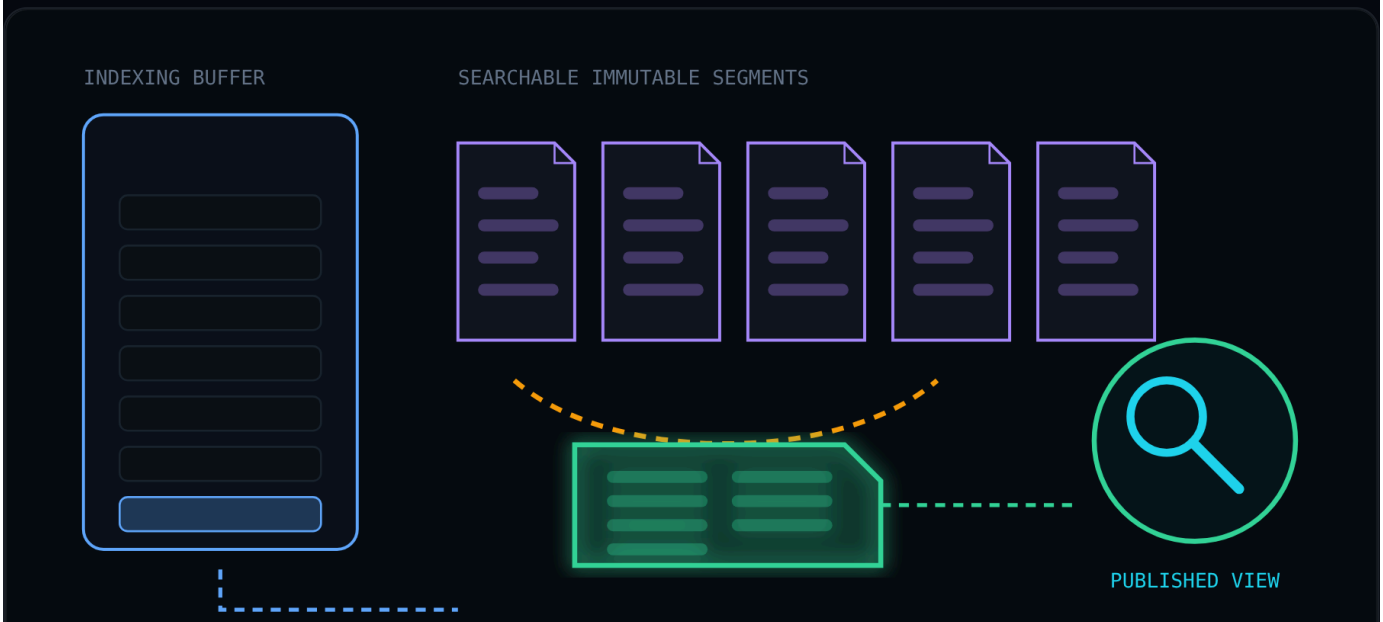
Skip this check

D4 takes the first slot because this fixture gives it evidence for both terms. Only the small top-hit lists traveled so far, although their global merge still sits on every search’s critical path and grows with shard fan-out.

Only after the winners are known does the fetch phase ask D4’s shard for its source and stored fields. Losing document bodies remain where they are, so the larger payload follows final IDs rather than every candidate.

Our 150-millisecond budget must cover routing, the slowest selected shard, the global merge and winner fetching. It also searches only the segment view each shard has published, which makes the five-second freshness promise the next unresolved cost.

Balance freshness and merge pressure



08 BALANCE FRESHNESS AND MERGE PRESSURE

Inside one shard, the view we just searched consists of immutable segment files. New postings cannot alter those files, so they collect in an indexing buffer while current readers continue using the already published view.

A refresh writes the buffered work as another searchable segment and opens a reader that includes it. Refreshing sooner shortens the wait for new documents, but it also creates small segment files more often.

After enough refreshes, one shard contains a growing row of immutable files, and each search has more pieces to cross. Which background operation prevents that row from growing forever?

PAUSE AND PREDICT

What reduces many small immutable segments?

- Background merging
- More replicas
- A new query analyzer

Skip this check

A merge rewrites several small segments into one larger file. Existing readers keep the old published view during that work, adopt the replacement when it is ready, and only then release the old files.

The handoff preserves search availability, but the rewrite still consumes CPU and disk bandwidth. Delaying it leaves more segments for queries, so segment count, merge I/O and p95 latency have to be interpreted together.

Refresh controls how soon a write becomes visible; merging controls whether that pace remains affordable. Indexing and search already share the same disks with both jobs before a failed node asks those disks to rebuild whole shard copies.

STAGE 9 OF 11

Recover and rebalance after failure

09 RECOVER AND REBALANCE AFTER FAILURE

To see the extra work caused by failure, this frame expands from one shard's segments to primary and replica copies spread across cluster nodes. A replica is not passive backup media: it can serve searches as well as preserve shard data.

Node A now leaves with P0 and R1. Every document still has a surviving copy, so requests can continue, but the cluster has lost two copies and the primary ownership that happened to live on that node.

R0 takes primary duty for shard 0, and requests can follow it immediately. Promotion restores an active owner; it does not create a replacement copy, so the available cluster is more exposed than it was before the failure.

A replacement node joins and can receive the missing shard files from surviving peers. If every recovery stream takes disk and network as fast as possible, what happens to the searches still serving users?

PAUSE AND PREDICT

Why should shard recovery concurrency be bounded?

- Protect serving capacity
- Rename shard IDs
- Change BM25 scores

Skip this check

The cluster allows two copy streams while live requests retain part of each machine. Redundancy returns less quickly than an unrestricted rush, but recovery is less likely to turn one node loss into a second outage through resource starvation.

When the streams finish, every shard has two copies again and allocation can spread load across four nodes. Disk watermarks, placement rules and serving latency still constrain how aggressively those copies may move.

Replicas kept the service alive, peer recovery rebuilt the missing copies, and snapshots cover different events such as deletion, corruption or wider cluster loss. None of that repair work receives private hardware; it competes on the same nodes as ordinary traffic.

Expose bottlenecks and tradeoffs

10 EXPOSE BOTTLENECKS AND TRADEOFFS

Recovery exposes the last false boundary: background work does not run on separate hardware. This frame puts bulk writes, searches and merges onto the same data-node CPU, heap, disk and network, with whole-shard recovery joining them after a failure.

Bulk ingestion occupies request memory, write threads, translog work and disk bandwidth. Larger batches can reduce overhead per document, but oversized batches hold more memory and make slow-tail behavior harder to contain.

Search adds matching, scoring, aggregation, coordination and fetch to those same resource rails. More shards can run pieces in parallel, yet each shard also adds setup, another queue and more state for the coordinator to merge.

Merges and recovery then collide with both foreground paths, so a tuning change can improve its own operation while hurting the service around it. What kind of test reveals that complete effect?

PAUSE AND PREDICT

How should the final capacity choice be validated?

- Mixed-load benchmark
- One empty query
- A fixed shard slogan

Skip this check

A mixed-load benchmark runs writes, searches, merges and recovery together. More shards buy parallel work with coordination and recovery; replicas buy read paths with disk and copied writes; faster refreshes and larger bulks move costs just as directly.

So the answer to our original 100 GiB question is a measured operating range, not one magic setting. Tail latency, queue depth, rejections, JVM pressure, disk watermarks, segment count, merge time and recovery throughput reveal whether the whole workload still fits.

STAGE 11 OF 11

Recap

11 RECAP

The first break in the single-disk picture was time: scanning 100 GiB at the illustrative rate took 204.8 seconds, while the service owed 250 QPS and p95 within 150 milliseconds. Size could not describe the workload.

Escaping that scan required stable term keys. Mapping gave every field its job, and compatible analysis let blue and axolotl reach D4 without turning its exact ID or publication date into language tokens.

Those keys made the inverted view possible. Ordered posting lists met at D4 without rereading source blocks, while frequencies and positions kept the evidence needed for BM25 scoring and phrase checks.

The shortcut then presented its bill. In this fixture, 100 GiB became 135 GiB of primary index data, 270 GiB with one replica and about 416 GiB at the disk target; bounded bulks kept write overload from growing without limit.

Paying for replica bytes still did not guarantee survival. P0 and R0 disappeared together inside one zone, but separating them left a live copy of every shard when that zone failed.

Distribution preserved copies but removed the one place that could rank everything. The coordinator therefore reached one eligible copy per shard, merged local BM25 results, and fetched stored fields only after D4 became the global winner.

That search saw only published, immutable segments. Refresh made buffered writes visible by adding new segments, while merging rewrote smaller files so the freshness promise did not leave permanent work on every query.

A failed node added that same kind of background pressure when the cluster was already weaker. Replica promotion restored ownership quickly; bounded peer recovery then rebuilt the lost copies without taking every disk and network resource.

This is why every useful knob moves a cost as well as a benefit. Shards, replicas, refresh cadence and bulk size can add parallelism, read paths, freshness or throughput while spending coordination, memory, disk, network or recovery budget.

The cluster was not chosen in advance and decorated with nine mechanisms. Each success exposed the next limit: term keys created index data, index data needed copies, copies needed placement, and serving had to leave room for visibility and repair.

A defensible OpenSearch design makes that chain testable. It states workload assumptions, shows the capacity arithmetic, explains what survives failure and names the signals that could prove the plan wrong; a memorized shard count does none of those things.