READ MODE · ANSWER FIRST

How Cursor Origin works technically with Continuity and Git

See how Cursor Origin uses local Git repositories, an S3-backed WAL, CAS, gossip, read repair and shared compaction.

CHEAT SHEET · 7 ESSENTIAL IDEAS

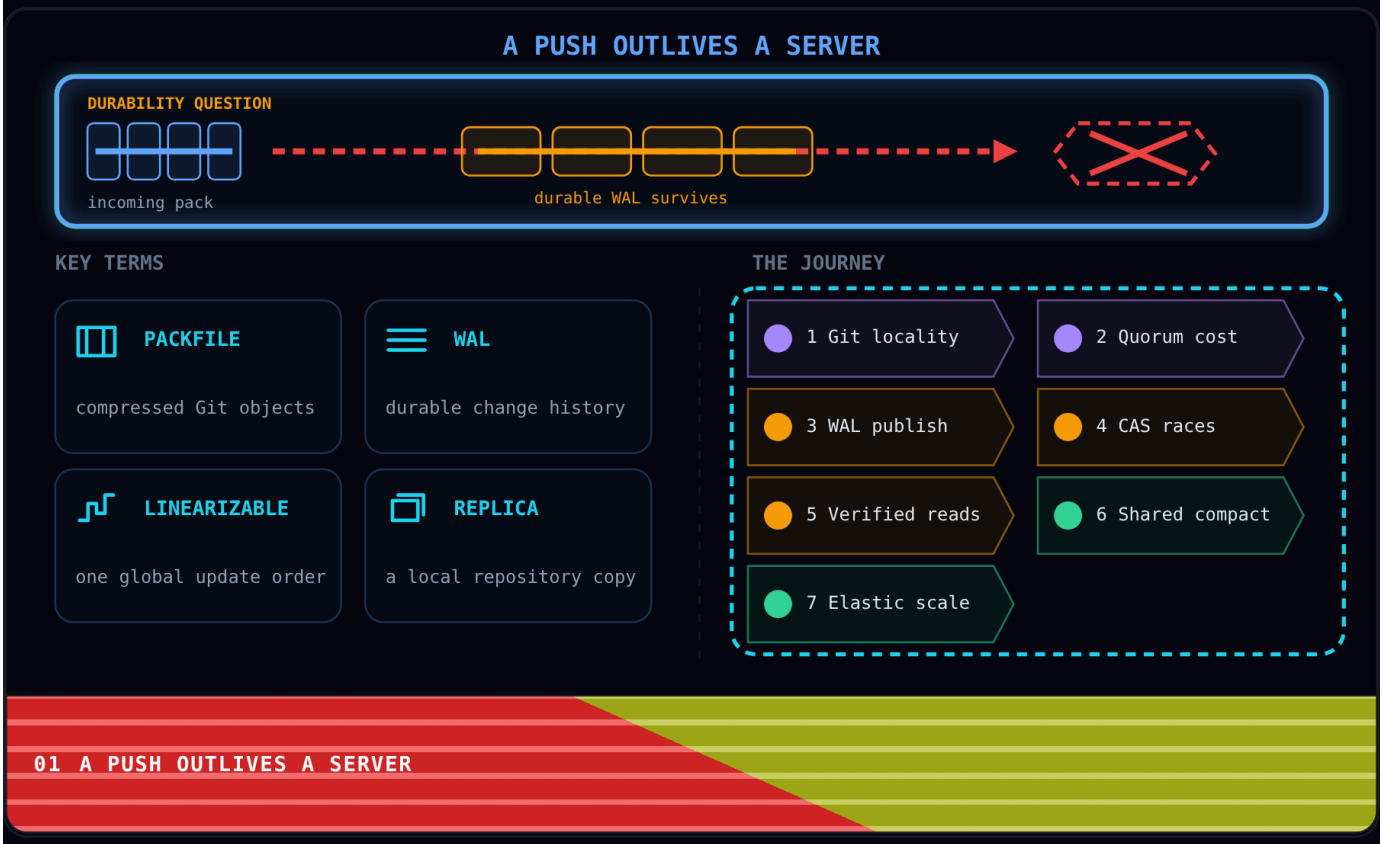
The whole story in 7 lines

Cursor Origin keeps Git fast on replaceable local disks while an object-store WAL owns durability, ordering and recovery.

1. Git stays fast on local NVMe because object graphs and delta packs create dependent random reads.
2. Replica-wide commit phases make three copies costly for tiny repos and slow larger replica groups.
3. A push becomes durable in the WAL before one index update publishes it and permits acknowledgement.
4. CAS serializes competing index updates, so a preferred server improves speed without owning correctness.
5. Gossip accelerates catch-up, while an ETag check prevents a stale replica from serving old state.
6. One primary repacks once, then followers trade download bandwidth for repeated compaction CPU.
7. Durable object storage lets idle repos have no warm copy and hot repos add many read replicas.

STAGE 1 OF 9

A Push Outlives a Server



Hold on to the instant after the push succeeds. The blue pack reached a server with a fast local Git repository, and then that server vanished. If success meant only “written to this disk,” the commit vanished with it.

Four ideas let us ask what success meant instead. A packfile carries Git objects. A write-ahead log, or WAL, preserves changes outside the server. One ordered index decides which WAL entries are published, and replicas are local repositories rebuilt from that history.

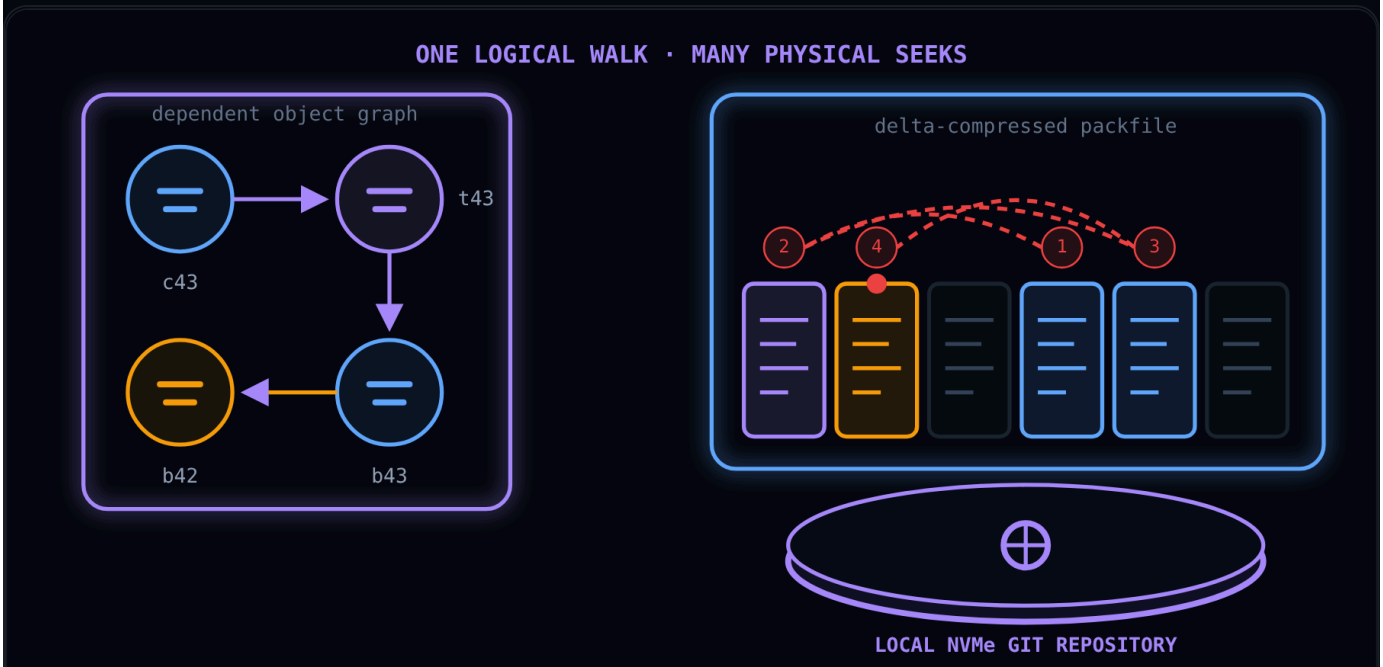
We will keep following the accepted pack. First we need to understand why Git still wants a local disk. Then we can separate durable bytes from a visible push, let two servers race to publish, and make a stale replica prove that it is current.

The server is now gone from the picture, while the amber WAL remains. Surviving bytes are a start, but recovery must also preserve which pushes became visible and in what order. That is the harder promise we are going to test.

It may seem easier to avoid fragile local repositories altogether and distribute Git’s data directly. Before accepting Cursor’s split between local speed and durable history, let us see why that tempting design runs into Git itself.

STAGE 2 OF 9

Why Git Wants Locality



02 WHY GIT WANTS LOCALITY

The tempting replacement for a lost disk is a distributed key-value store: use each Git object ID as a key and fetch the object from anywhere. Our illustrative walk begins with one commit object, `c43`. Reading that object is not the end of the operation. It reveals where the walk must go next.

The commit points to tree `t43`, and that tree points to blob `b43`. Git cannot request this whole chain in advance because each object contains the identifier of the next one. A history or directory walk therefore unfolds one dependent lookup at a time.

The logical chain is only half the problem. Its objects occupy scattered positions in a compressed packfile, and a blob may point again to a delta base. If each newly discovered object lives behind a network request, which cost repeats at every hop?

PAUSE AND PREDICT

What dominates a remote object-by-object Git walk?

- **Parallel bandwidth**
- **Dependent round trips**
- **Hash computation**

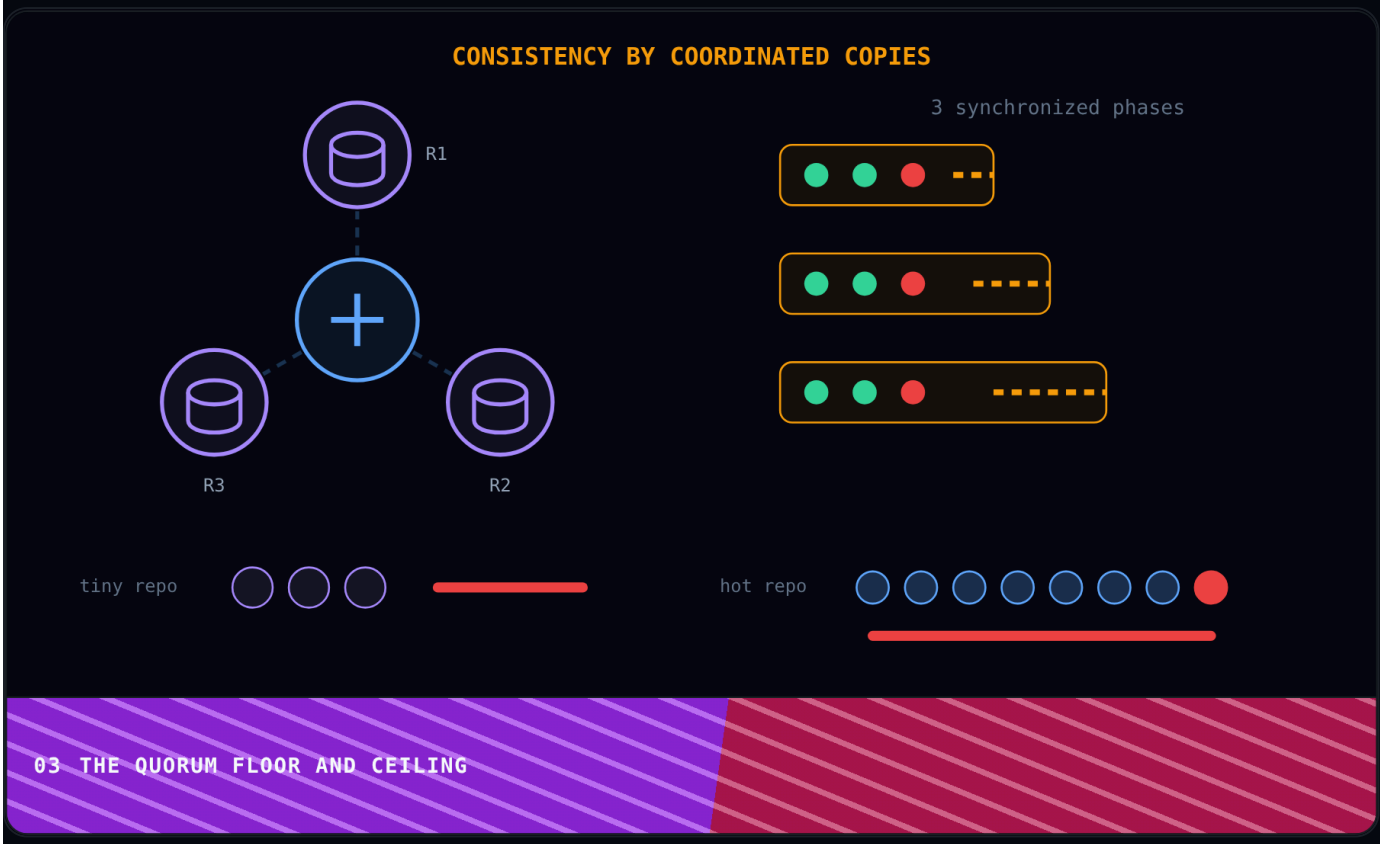
Skip this check

The walk reaches `c43`, `t43`, `b43` and its base in sequence, so network latency would accumulate along the chain. A local NVMe repository turns those dependent graph and packfile seeks into local

reads. Cursor kept this part of the older design because ordinary Git is already good at it.

So the local repository is not an accidental cache we can casually replace with remote object lookups. It is what keeps normal Git operations fast. The obvious way to make that disk survive failure is to keep several synchronized local copies. That works, but it changes what every push must wait for.

The Quorum Floor and Ceiling



03 THE QUORUM FLOOR AND CEILING

Cursor’s earlier Spokes design took that direct route. An orchestrator fanned each push out to several complete Git repositories. The three disks shown here are the minimum replica set in our worked view. Because those copies were the source of truth, they had to agree.

Spokes synchronized the small reference transaction with three-phase commit: vote, pre-commit and commit. The pack itself could fan out, but the push was not accepted until the coordinated reference update had passed through those rounds.

Three copies give an idle repository a costly floor. A busy repository wants more copies for read traffic, yet every added participant gives each synchronized phase another machine to wait on. What happens to push latency as that group grows?

PAUSE AND PREDICT

What happens as coordinated replica count grows?

- **Pushes get faster**
- **Slow-tail exposure grows**
- **Nothing changes**

Skip this check

The same consistency scheme pinches from both sides. A tiny repository keeps three mostly idle copies, while a hot repository’s push moves at the pace of the slow tail in a larger coordinated group. More read

capacity makes the write path harder.

There is an operational consequence too: when local copies are the durable truth, every copy is precious and its location must be tracked and repaired. Continuity keeps the fast local Git repository, but makes a different record authoritative. Our accepted pack is ready to enter it.

STAGE 4 OF 9

Durable Before Visible

To keep Git local without making the disk authoritative, Continuity starts with the same thing an ordinary client already sends: a Git packfile. No special client-side storage protocol is needed. The change begins after this pack reaches an Origin server.

The incoming pack now takes two paths at once. The server writes it into the local Git repository for fast access and uploads it as a WAL entry to S3-compatible object storage. Once that upload persists, the bytes can outlive this server—but no branch points to them yet.

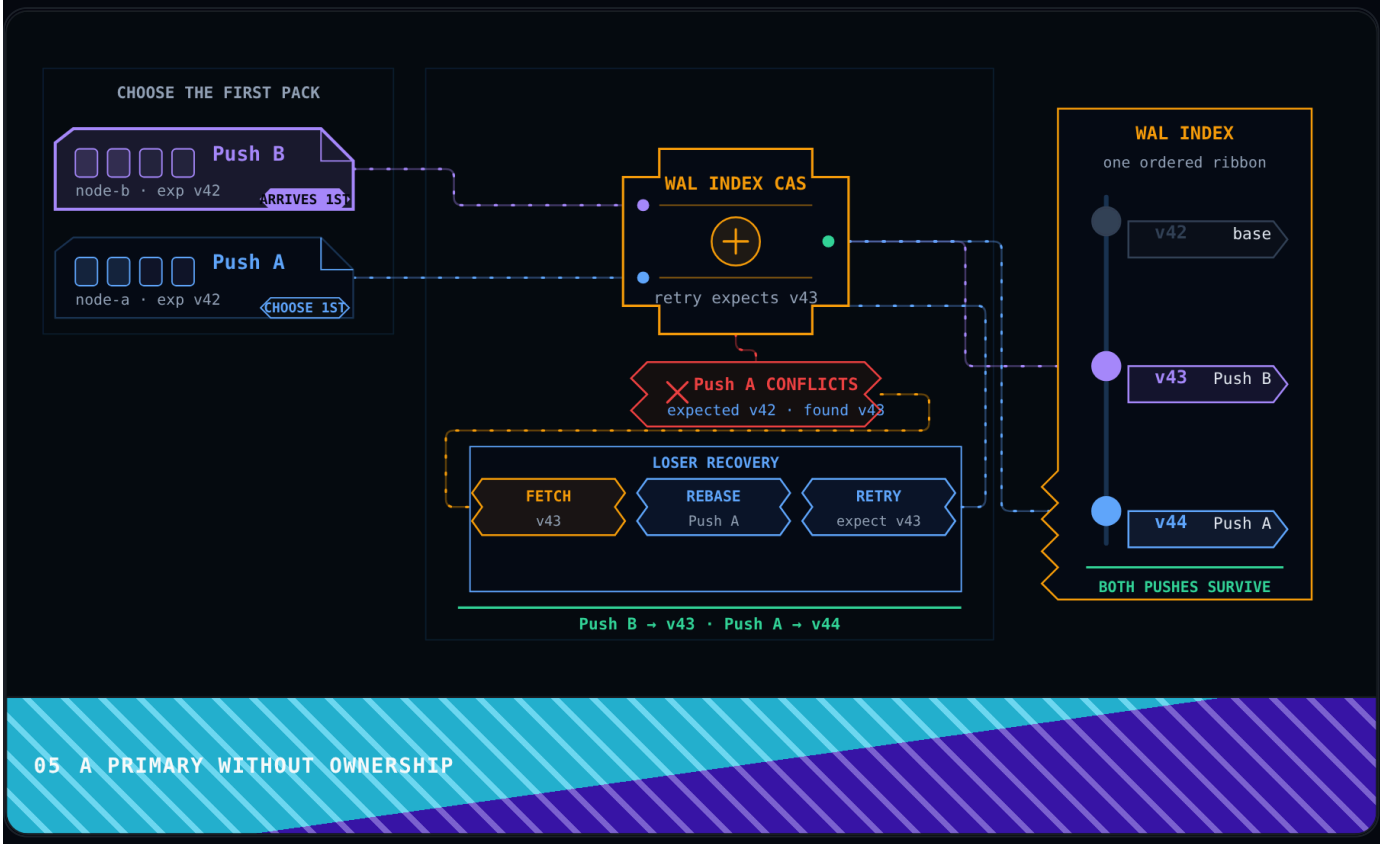
Visibility begins with a prepared Git reference transaction on the local repository. Preparation checks that the branch still has the expected old value and holds the update ready. The durable pack is therefore not confused with a published commit.

The shared WAL index advances to the illustrative version `v43` and records the entry in the published order. Only after that update does Origin acknowledge the push. A replacement server can now recover not just the pack bytes, but the fact that this push became visible.

We have separated two moments that first looked identical: the WAL entry makes the push durable, and the index makes it visible. That single index also becomes the place where concurrent pushes must agree, even when they arrive at different replaceable servers.

STAGE 5 OF 9

A Primary Without Ownership



05 A PRIMARY WITHOUT OWNERSHIP

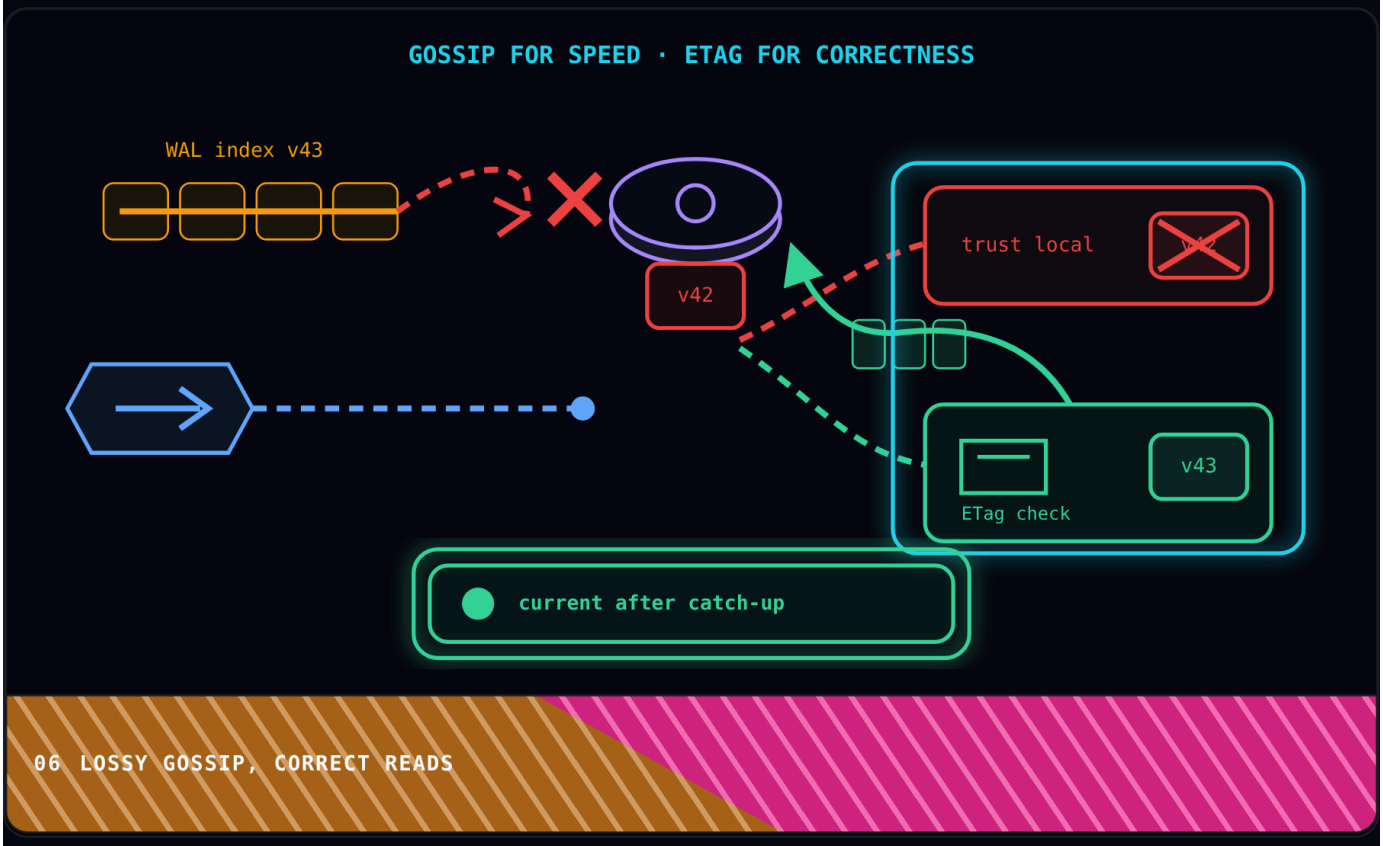
The shared index now faces two illustrative pushes, A and B, both expecting `v42`. Choose which pack arrives first. CAS publishes it as `v43`. The loser fetches that index, rebases and retries as `v44`. A preferred server improves speed, but CAS owns correctness.

STAGE 6 OF 9

Lossy Gossip, Correct Reads

★ If you remember one thing · Origin can make gossip lossy because every read validates replica freshness against the durable WAL before serving.

Try it in Watch mode. Find the counterfactual combination that can serve stale Git state.



06 LOSSY GOSSIP, CORRECT READS

CAS has established a published `v43`. The other local repositories still need to learn about it. Continuity sends small UDP gossip packets as hints. A hint carries enough metadata for a replica to catch up from S3, not the repository contents themselves.

Now the cyan hint is dropped. The durable WAL index remains at illustrative version `v43`, while this local repository still holds `v42`. If Origin simply trusted the fast disk, the push we worked so hard to publish could disappear from a later fetch.

A client fetch reaches that stale replica. The upper lane serves local state directly. The lower lane conditionally checks the WAL index with the replica's last ETag. Which lane can discover and install `v43` before replying?

PAUSE AND PREDICT

Which read lane returns the newest version after gossip is dropped?

Trust local

ETag check

Both lanes

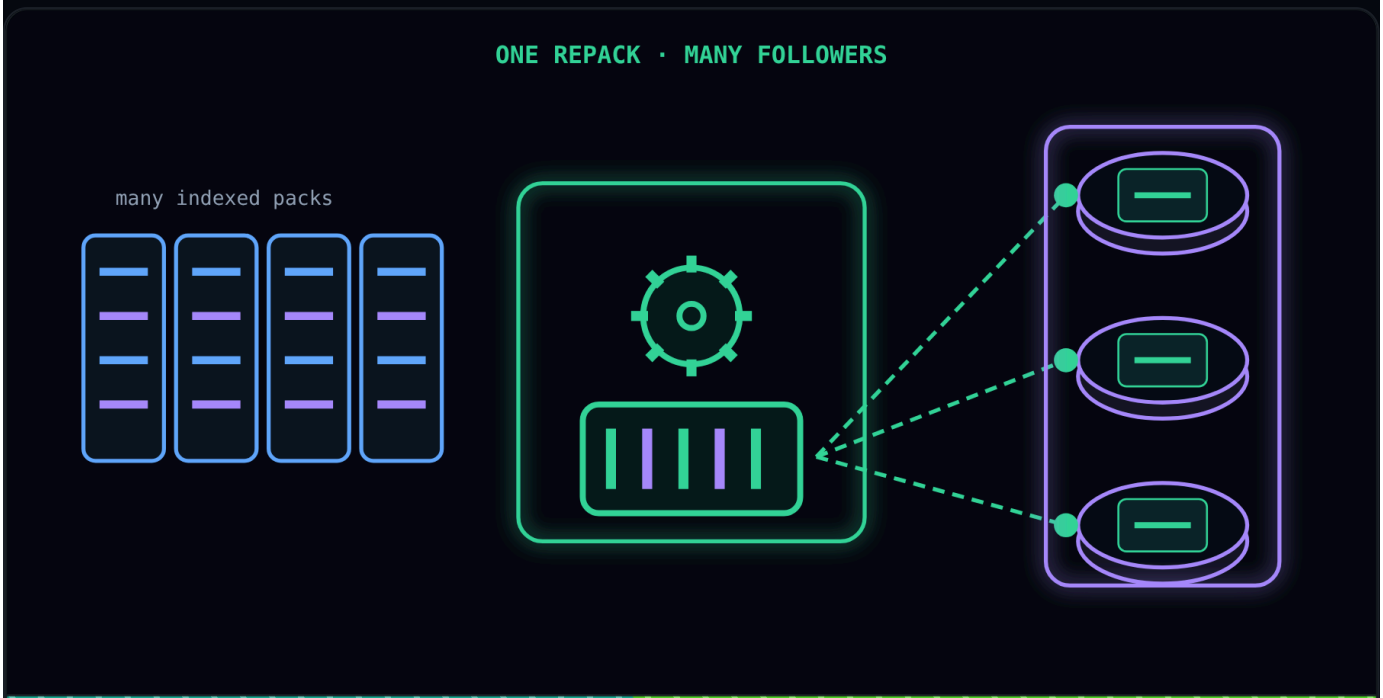
`Skip this check`

The ETag check returns the newer index, catches the repository up to `v43`, and only then serves the fetch. The lost UDP packet changes how much repair work the read performs. It does not change the version the client is allowed to see.

Switch the Read guard between ETag check and Trust local. With gossip dropped, compare versions and find the choice that serves stale state.

This is why unreliable gossip can remain an optimization: every read verifies freshness against the durable index. But recovery cannot grow into an endless replay, and repeated pushes also leave more packfiles on each local repository. The next cost is maintenance, not correctness.

Compact Once, Download Many



07 COMPACT ONCE, DOWNLOAD MANY

In this worked trace, repeated pushes have left four separate packfiles in one local Git repository. Each pack has its own index. As that collection grows, finding objects and rebuilding a repository requires more work, so Git eventually has to combine the fragments.

Repacking is CPU-heavy. If every synchronized copy performs it independently, the same computation runs on every follower and can compete with the Git operations those machines are meant to serve.

Continuity already has one durable history that every replica follows. If a completed repack can be published into that history like any other change, which machines still need to spend CPU producing it?

PAUSE AND PREDICT

Who performs the expensive repack in Continuity?

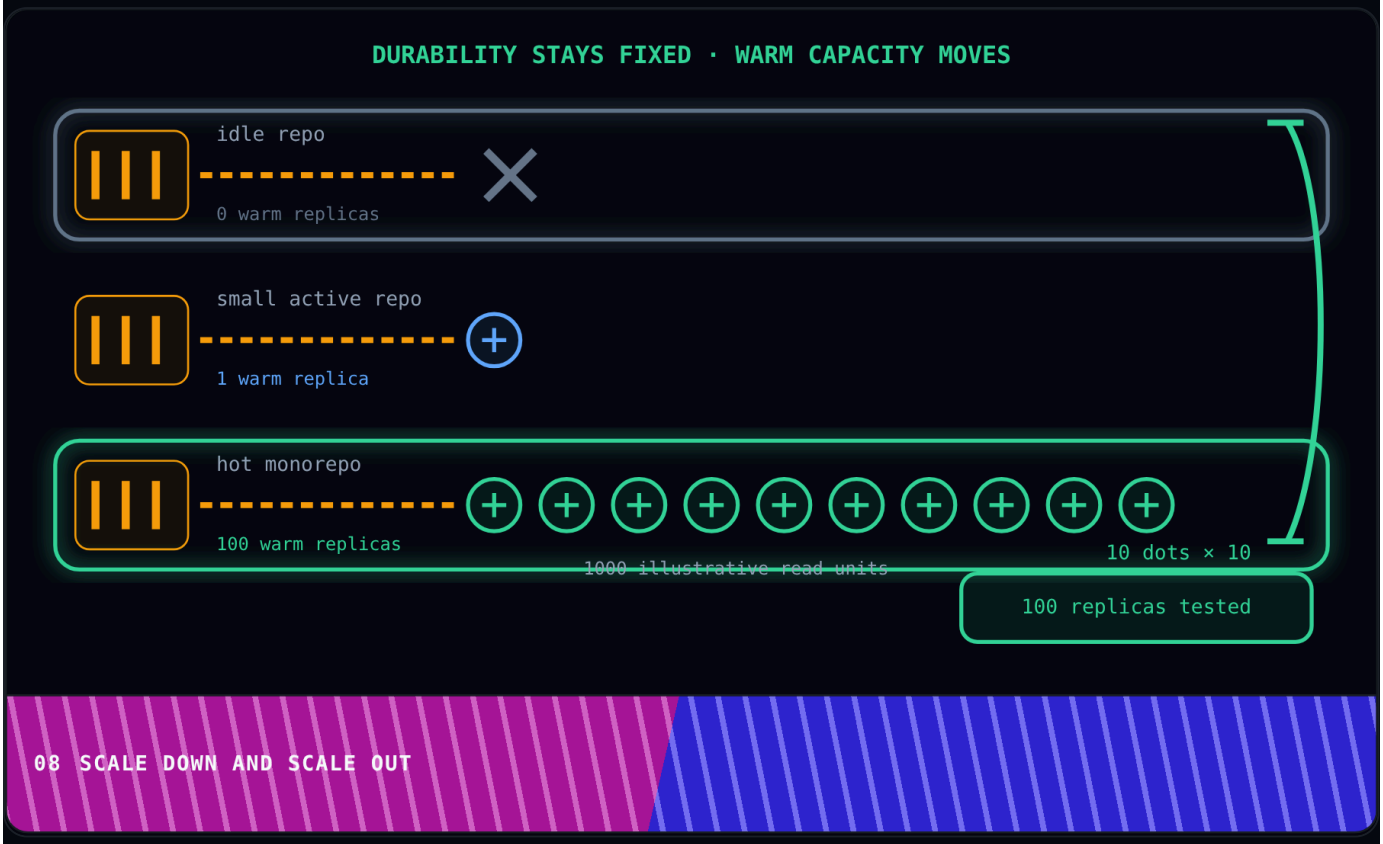
- Every replica
- One primary
- Each Git client

Skip this check

One primary merges the four packs and records the compacted result in the WAL. The followers download those bytes instead of running their own repacks. Continuity deliberately trades shared bandwidth for repeated CPU work.

The WAL now preserves both published pushes and the compacted state that makes restoration practical. A local repository can be discarded and rebuilt without losing either correctness or all prior maintenance work. That removes the durability reason to keep a fixed number of warm copies.

Scale Down and Scale Out



08 SCALE DOWN AND SCALE OUT

Once durable history can rebuild a compacted repository, an idle repository no longer needs a disk kept warm for safety. The first row therefore has zero warm replicas. Its code has not vanished. Only the disposable local copy has.

A small active repository can keep one warm replica to avoid rebuilding on every access. If that server disappears, rendezvous hashing selects another expected node and the repository materializes again from the WAL.

A hot monorepo has the opposite problem: clones, fetches, CI and Origin RPCs need more read capacity. It can add many current local repositories, while the WAL index still supplies one published write order.

The same durability mechanism now supports zero, one or many warm copies according to demand. Cursor reports synthetic tests with linear read scaling through 100 replicas and no reduction in push throughput. This explainer has not independently reproduced that result.

Cursor also reports up to 120 pushes per second on S3 Standard and more than 300 on S3 Express One Zone. Those are Cursor’s published benchmarks, not the meaning of our illustrative dots. We can now reconnect the whole answer to the vanished-server question.

STAGE 9 OF 9

The Whole Origin Storage Loop

09 THE WHOLE ORIGIN STORAGE LOOP

Place the durable WAL at the center and return to the reason local repositories remain around it. Git discovers object pointers one by one and follows scattered packfile data, so ordinary Git operations stay fast on local NVMe.

Making several local copies the source of truth preserved consistency in Spokes, but created a three-copy floor and a slow-tail ceiling. It also made every placed repository important operational state.

Continuity moved that durable responsibility into an object-store WAL. An incoming pack persisted first. A prepared reference transaction and versioned index update then made it visible in one recoverable order.

That versioned index gave concurrent writers somewhere precise to disagree. CAS accepted one expected version, while the conflicting push fetched the winner, rebased and retried instead of being lost.

The same index settled the stale-read problem. Gossip could be dropped because a conditional ETag check forced an old local repository to catch up before it served a fetch.

The WAL also carried maintenance forward. One primary repacked the Git data, published the result, and let followers download it rather than repeat the expensive CPU work.

Once any warm copy could be rebuilt from that history, replica count stopped being a durability requirement. Idle repositories could have zero warm copies, while hot repositories could add readers as demand grew.

Now the vanished server is no longer a paradox. Local Git supplies the speed Git’s data layout needs, but the WAL and its ordered index remember every acknowledged state. A server may disappear because no individual local repository has to be the truth.