SyncNotes
A Consistency-Oriented Replicated Store for Collaborative Notes

Final Report for the Distributed Systems Course A.Y. 2025/2026

Marzia De Maina
marzia.demaina@studio.unibo.it

September 15, 2026

SyncNotes is a collaborative note-taking system built around a three-tier client-gateway-cluster architecture. A set of shared plain-text notes lives on a cluster of three interchangeable server replicas, the sole source of truth; a lightweight, stateless gateway is the cluster's single logical entry point, so clients never learn how many replicas exist, which one is serving them, or that a failover happened. The client itself is a full-screen terminal UI requiring no knowledge of commands, note names or file paths, so a non-technical user can browse, edit, create and delete notes with nothing but the arrow keys.

The system is deliberately **CP** in CAP terms. Writes are ordered by a Raft replicated log and applied identically on every replica, so all clients see the same version of every note, and a replica that loses quorum during a network partition refuses writes rather than diverging. Concurrent edits are reconciled by a three-way merge that runs inside the replicated operation itself: disjoint edits are combined automatically, and a directly conflicting line is resolved deterministically in the server's favour, with the losing client told exactly what happened. On top of consensus, a background reconciliation loop lets a lagging or corrupted replica repair itself from a healthy peer, closing the one gap consensus cannot see: storage-level corruption that never went through the log.

The whole cluster, plus the gateway, runs as Docker containers wired by a single compose file; the client runs on the host. Validation combines an automated pytest suite, an end-to-end merge demo against a real cluster, and chaos scripts that kill the leader, partition the network, corrupt a replica's storage and restart a node, asserting the system recovers exactly as the requirements demand.

Disclaimer

During the preparation of this work, the author used a large language model (Anthropic Claude) to rewrite some sentences of this report for clearer English syntax and to review the wording of technical explanations, to help debug code and to generate the automated test suite. Every other part of the artifact is the author's own work.

After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the final report and of the artifact.

1 Concept

1.1 Product type

The product is a **collaborative note-taking application** composed of a client, a gateway and a server cluster. The client is delivered as a *full-screen terminal UI*, a curses application. The **gateway** is a web service acting as the single logical entry point of the cluster, which is in turn a **server cluster** of interchangeable replicas (three in the reference deployment), each a web service backed by its own embedded database.

1.2 Use case description

- **What the software does**

  Users keep a set of shared plain-text notes: a user browses the list of notes on the cluster, opens one, edits it locally and saves it back and the cluster merges the edit against the current authoritative content and returns the merged result as the new starting point for the next edit. Users can also create and delete notes: a deletion is visible to everybody and the note list and any already-opened note are kept up to date automatically while the UI is open.

- **Where the users are**

  Users sit at separate machines on a shared network (a LAN, a VPN, or the same host during development), each running a client process that reaches the cluster through the gateway's address. The cluster itself is a group of machines (here, containers) co-located on one network segment.

- **When and how frequently they interact with the system**

  Interaction is session-based and bursty: a user opens the UI, works on one or a few notes for some minutes with frequent saves and closes it. Several users may have a session open at once and may edit the same note: their saves are validated one at a time, in arrival order, and between explicit saves each open client polls the note list in the background (a server-sent-events stream, with a fallback poll every few seconds).

- **How they interact with the system and on which devices**

Interaction happens entirely through a terminal on a desktop or laptop: the full-screen UI is keyboard-driven (arrow keys to move, `Enter` to open, `n` to create, `d` to delete, `r` to refresh, `q` to quit).

- **The data the system stores, and where**

  The system stores the notes themselves, for each note: its name, its version number, current text content, a content hash SHA-256, a last-modified timestamp and a deleted tombstone flag, plus the full text of every past version and a ledger of already-applied operation ids (for idempotent retries). This data is replicated on every server replica and the client keeps a small hidden sidecar file next to each downloaded note recording which version it was based on.

- **There are multiple roles**

  There is a single human role, the *user*, who can read and write every note with no per-note ownership or access control, while internally the servers play two transient roles assigned by the consensus protocol: one *leader* at a time, which owns writes and the remaining *followers*, invisible to users and reassigned automatically on failure.

1.3 Why distribution is needed

- **Geographically distributed environments**

  Distribution is *not* motivated by geographic dispersion at planetary scale, by computational speed-up or by handling a very large user population.

- **Resource sharing**

  The core purpose is a set of notes shared by several users who edit them concurrently from different machines. A shared mutable resource, reachable by many independent clients, is inherently a distributed-system problem: the notes cannot live only on one user's machine.

- **Fault tolerance**

  Fault tolerance requires more than one server holding the authoritative state: the gateway reroutes client to the current leader when a replica goes offline and replicas reconcile against their peers to recover from local corruption, both impossible with a single server.

- **Availability of the entry point**

  Clients reach the cluster through one gateway endpoint instead of a specific server, so the gateway must keep discovering and forwarding to whichever replica is currently leader, hiding crashes and leader changes behind that single address. This indirection only makes sense in a distributed setting.

2 Requirements Elicitation and Analysis

Requirements are numbered and grouped into functional (**F**), non-functional (**N**) and implementation (**I**). Each carries an acceptance criterion.

2.1 Glossary

- `Note`: a named, shared plain-text file managed by the cluster.
- `Authoritative content`: the version of a note held and agreed upon by the server cluster; the only version visible to all clients.
- `Provisional copy`: a note as downloaded and possibly edited on a client.
- `Base version`: the note version a provisional copy was derived from; recorded client-side and sent back on upload as the common ancestor for merging.
- `Sidecar`: the small hidden file, next to a locally-downloaded note, that records its base version; where the base version actually lives on the client.
- `Quorum`: a strict majority of replicas; required to commit a write.
- `Tombstone`: a row that marks a note as deleted while remaining present, so the deletion can propagate.
- `Reconciliation`: the background process by which a replica compares its state against a peer and pulls whatever it is missing, stale on, or corrupted on.

2.2 Requirements

2.2.1 Functional requirements

- **F1: List available notes**

  A user must be able to obtain the list of notes that currently exist on the cluster, with at least each note's name and version. Deleted notes must not appear.

  Acceptance criterion: after uploading notes a and b and deleting b, a list request returns exactly $\{a\}$ with a's current version.

- **F2: Download a note**

  A user must be able to fetch the current authoritative content of the notes, including their name and version. Fetching a note that has been deleted must be distinguishable from fetching a live note, since the server keeps a tombstone rather than removing the record.

  Acceptance criterion: fetching an existing note returns its exact authoritative content and version; fetching a name that never existed and fetching a deleted note both return HTTP 404 (the gateway turns a replica's tombstone into a 404 for every client, even though a replica's own HTTP surface keeps the tombstone visible internally with a `deleted` flag for reconciliation's sake).

- **F3: Save a note with conflict-aware merging**

  A user must be able to edit the content of a note, together with the base version it was edited from. The cluster must combine the edit with any changes that occurred since that base version, according to a deterministic rule in which the server's content always wins on directly conflicting lines and return the merged result. The user must be told whether their edit was applied cleanly, merged with other changes, or partly dropped due to a conflict.

  Acceptance criterion: more than one client editing *disjoint* lines of the same base see their edits survive; clients editing the *same* line result in the server's version being kept and the last client being told its change to that line was dropped.

- **F4: Create a note**

  A user must be able to create a new, empty note under a chosen name. Creating a name that already exists must not overwrite the existing note.

  Acceptance criterion: creating `new.md` makes it appear in the list at version 1 with empty content; creating a name that already exists leaves the existing content untouched and reports that nothing was created.

- **F5: Delete a note for everyone**

  A user must be able to delete a note. After deletion the note must disappear from every client's list. A later creation of the same name must be allowed and must start a fresh note.

  Acceptance criterion: after one client deletes a note, every other client's list no longer contains it; re-creating the name yields a new and empty note.

- **F6: Automatic propagation to connected clients**

  While a client session is open, changes made through any client (including changes applied by internal reconciliation) must become visible to the other open sessions without a manual refresh, within a small bounded delay.

  Acceptance criterion: with two UI sessions open, an upload or a deletion performed in one is reflected in the other's note list within a few seconds; already-downloaded local copies of a changed note are refreshed automatically unless that note is currently being edited.

- **F7: Usable by a non-technical third party**

  The system must offer an interface that requires no knowledge of commands, note names, file paths or the cluster. Note names entered by the user must be validated with an understandable message on rejection.

  Acceptance criterion: a person can launch one command, browse notes with arrow keys, open and edit a note, create and delete notes and quit, without ever typing a note name or path except when naming a new note; an invalid new name (empty, containing a path separator, a leading dot, reserved characters) is rejected with a specific reason.

2.2.2 Non-functional requirements

- **N1: Location, replication and failure transparency**

  Clients must interact with one fixed logical endpoint and must not need to know how many replicas exist, which replica serves a request, which replica is the leader, or that a failover happened.

  Acceptance criterion: no client configuration references an individual replica; killing the replica a client was implicitly using does not require the user to change any address or restart against a different one.

- **N2: Strong consistency of authoritative state**

  All clients must observe the same content and version for a note at any point where the cluster has a leader. A successful write must be immediately visible to a subsequent read.

  Acceptance criterion: a read issued after a successful write, through the gateway, returns the just-written content; the three replicas' own databases converge to identical tuples.

- **N3: Total order of writes**

  Concurrent uploads must be applied one at a time in a single global total order [Lamport, 2019], FIFO queue, and the merge decision for each upload must be a deterministic function of the ordered history, not of which replica evaluated it.

  Acceptance criterion: replaying the same sequence of uploads yields the same final content on every replica; no upload is lost when several arrive at once.

- **N4: Tolerance of a single replica failure, including the leader**

  The cluster must keep serving reads and writes when any one replica (a follower *or* the current leader) crashes: reconfiguration and no loss of already committed data.

  Acceptance criterion: having the current leader killed is followed by a new leader getting chosen among the survivors and by the gateway routing a subsequent write to it successfully; the previously committed content is intact.

- **N5: No split brain under partition**

  If the network partitions the replicas, at most the majority side may accept writes. A minority replica must not accept writes and must not present itself as authoritative.

  Acceptance criterion: isolating one replica of three leaves the majority serving normally, while the isolated replica reports that it has lost quorum; after the partition heals, the isolated replica converges to the content produced by the majority.

- **N6: Data integrity and corruption self-healing**

  Silent corruption of a replica's stored content must be detected by that replica and repaired from a healthy peer.

Acceptance criterion: overwriting a follower's stored note content directly in its database, leaving the recorded hash unchanged, is detected on the next reconciliation pass and the correct content is restored from the leader.

- **N7: Automatic rejoin after restart or address change**

  A replica that is stopped and restarted must rejoin the cluster on its own and catch up on writes it missed, without operator action.

  Acceptance criterion: force-recreating a follower container causes the remaining replicas to re-resolve its address and re-admit it and the recreated replica ends up holding every write committed while it was away.

- **N8: Deletion convergence**

  A note deleted while a replica is offline must remain deleted after that replica rejoins; it must not be resurrected from the stale copy the offline replica still holds.

  Acceptance criterion: deleting a note while one replica is down and then restarting that replica leaves the note tombstoned on all three replicas, with the note absent from the user-facing list.

- **N9: Idempotent, retry-safe writes**

  Re-sending an upload or a deletion that may already have been applied must not apply it twice.

  Acceptance criterion: submitting the same upload twice with the same operation identifier bumps the version only once; the second submission returns the already-committed result.

- **N10: Responsiveness of the interactive client**

  The full-screen UI must stay responsive regardless of network activity and must never block indefinitely on a slow or dead cluster.

  Acceptance criterion: with the cluster stopped, the UI still redraws, still accepts `q` and shows a readable *cannot reach SyncNotes* status instead of hanging.

- **N11: Unavailability, not corruption, when quorum is lost entirely**

  If enough replicas fail that no side of the cluster retains a majority, the system must refuse writes rather than let a lone, non-quorate replica accept them as authoritative. Service must resume automatically, with no data loss, as soon as a majority becomes reachable again.

  Acceptance criterion: stopping the minimum number of replicas needed to break quorum leaves the survivor reporting `has_quorum=false` (`leader` itself is not a reliable signal here: pysyncobj leaves it pointing at the last known, now-dead leader until the survivor's own election timeout fires); a write submitted during this window never commits and eventually times out rather than succeeding; restarting the stopped replicas restores quorum and the gateway resumes serving writes with

all previously committed content intact. Verified directly, without Docker, by `server/tests/test_quorum.py`, which runs three real `NotesStore` (Raft) nodes in-process.

2.2.3 Implementation requirements

- **I1: Python for client and server**

  The client, the server and the gateway are all implemented in Python.

  Justification: Python is high-level socket, threading and HTTP abstractions suit a prototype-scale distributed system and a mature Raft library is available.

  Acceptance criterion: the whole codebase is Python; the server and gateway containers are pinned to CPython 3.12, and the client, run on the host rather than containerized, targets CPython 3.12+.

- **I2: A SQL database for persistence**

  Persistent state is to be stored in a SQL database.

  Justification: SQLite was chosen because it is embedded and needs no separate server process to configure and because giving each replica its own database file makes it visually obvious in a demo that the cluster is three independent stores converging to the same state, not one shared datastore behind the scenes.

  Acceptance criterion: each replica persists to its own SQLite file. A direct SQL inspection of the three files shows the same rows after convergence.

2.3 Relevant Distributed System Features

- **Transparency**

  The system must provide *access* transparency (a uniform HTTP interface), *location* transparency (clients hold only the gateway address), *replication* transparency (clients are unaware there are three copies) and *failure* transparency for the common case (a single replica or leader failure is masked by re-routing). *Migration* transparency also applies: a replica may come back at a different address. Full failure transparency is *not* promised: a total loss of quorum is surfaced to the user as a temporary error.

- **Consistency**

  The system is **CP**: it favours a single, agreed-upon state over availability of every replica [Gilbert and Lynch, 2002]. Writes are linearised through a Raft log [Ongaro and Ousterhout, 2014]. Reads are served only from the leader, so a client never reads state older than its own last successful write. Eventual convergence of a lagging or repaired replica is provided by reconciliation on top of consensus.

- **Fault tolerance and dependability**

Availability: the cluster tolerates one replica down (quorum). *Reliability*: committed writes survive crashes via the persisted Raft journal and the per-replica database. *Integrity*: a per-note content hash lets a replica detect corruption of its own storage and repair it. *Recoverability*: a restarted replica rejoins and catches up unattended.

- **Partition tolerance**

  In this deployment, partitions come from Docker's container networking rather than from geography: a replica can lose its connection to the others because of a network hiccup, a container restart, or a deliberate network disconnect during testing. The Raft implementation already handles this without extra logic in SyncNotes: a replica cut off from the other two can no longer collect the votes of a majority, so it cannot hold or acquire leadership and reports `has_quorum=false` through its health endpoint. The gateway only ever routes writes to a replica that reports itself as leader with quorum, so the isolated replica is structurally excluded from accepting writes while it is alone; the majority side keeps its leader and keeps serving normally. When the partition heals, the previously isolated replica rejoins the Raft group and reconciliation catches it up to the authoritative content, so no divergence survives.

- **Scalability**

  Both the Raft peer list on each server and the replica list the gateway polls are plain configuration, so growing the cluster to any N replicas is a deployment change, not a code change. What remains deliberately unaddressed is read scalability: reads are served only from the leader, never distributed to followers, so adding replicas improves fault tolerance and quorum size but not read throughput. For the target of deploying this project for academical purposes this trade-off keeps the consistency argument simple.

- **Resource sharing**

  The notes are the shared resource; coordination over concurrent writes to them is the core mechanism.

- **Openness, interoperability, heterogeneity**

  The client-gateway and gateway-replica interfaces are plain HTTP with JSON bodies, so an alternative client in another language could be written against the same contract. Internally, replicas are homogeneous.

- **Security**

  Every endpoint, client-gateway and gateway-replica alike, is plain HTTP with no authentication, no authorization and no encryption in transit; the design assumes a trusted network, which matches the *shared team notebook* use case running on a private segment among trusted team members.

- **Performance, concurrency, bandwidth: moderately relevant**

Notes are small text files and write rate is low (interactive editing), so consensus round-trips and full-content replication are affordable. Concurrency is real (multiple clients, multiple server threads) and is handled by the Raft single-writer model plus per-request database sessions.

- **Economy**

  The whole cluster runs as lightweight containers on a single host: SQLite and pysyncobj need no external infrastructure of their own, no separate database server, no message broker, no orchestration beyond `docker-compose`. Because growing the cluster is a configuration change rather than a code change, the cost of adding replicas scales linearly with N container instances, with no additional infrastructure to provision or license.

3 Design

3.1 Architecture

SyncNotes uses a **three-tier client-gateway-cluster** architecture (fig. 1).

- The **client** tier is a thick client: it holds provisional copies of notes, performs the local editing workflow, tracks the base version of each copy and renders the UI. It has no authority over state. Rendering a single interactive, keyboard-driven screen, rather than exposing raw commands or file paths, is what delivers F7.

- The **gateway** tier is a thin, stateless *reverse proxy* specialised for this cluster. Its only job is to present one address to clients and to forward each request to the current leader, re-discovering the leader and retrying when the target has changed or become unreachable. It also fans out change notifications to connected clients.

- The **cluster** tier is a *replicated state machine*. Every replica runs the same deterministic note-store logic over the same Raft log, so all replicas hold the same state. Exactly one replica is the leader; it is the only one that accepts writes and the only one the gateway reads from.

- **Why this style**

  The client-gateway split delivers N1 (transparency): clients bind to the gateway, never to a replica, so the replica set can change freely. The replicated-state-machine cluster delivers N2 and N3 (strong consistency, total order): consensus gives a single agreed order of writes and running the merge *inside* the replicated operation makes the outcome a pure function of that order. Leadership plus quorum delivers N4 and N5 (failure tolerance, no split brain). A separate stateless gateway, rather than teaching clients to find the leader themselves, keeps that failover logic in one place and keeps the client contract trivial.

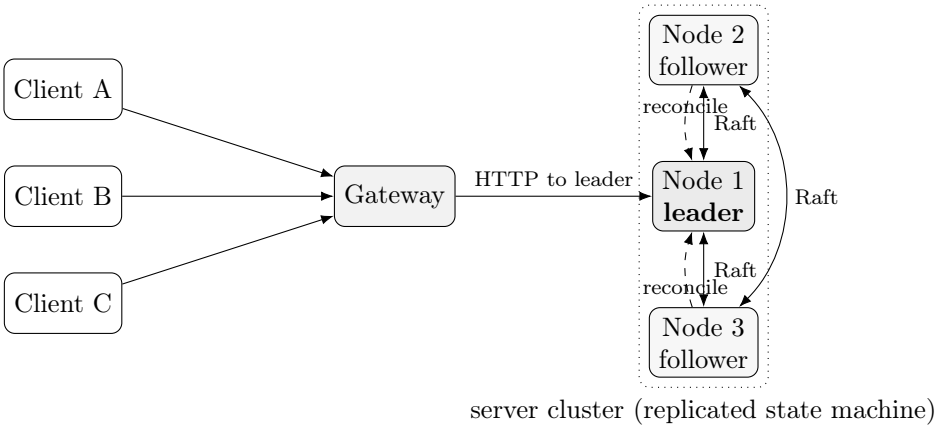

Figure 1: Three-tier architecture. Clients speak only to the gateway; the gateway speaks only to the current Raft leader; replicas keep their state machines aligned through the Raft log and, as a safety net, through peer reconciliation.

- **Alternatives considered**

  A leaderless, quorum-read/quorum-write design would improve availability but weakens the simple *server always wins, everyone sees one version* contract this design commits to (F3, N2). A single server with a hot standby would be simpler but fails N5's spirit and would not meet the minimum of three replicas needed to keep a majority quorum after one failure (N4). Consensus over a small static cluster is the least-effort way to get a defensible consistency story.

3.2 Infrastructure

- **Components and their multiplicity**

  The infrastructure comprises: *clients* $(0..N)$, one *gateway* service and a *server replica* service with a fixed multiplicity of three. Each replica embeds its own database; there is no external database server, no load balancer, no message broker, and no cache. The gateway is the only externally published endpoint.

- **Distribution over the network**

  In the reference deployment all server replicas and the gateway are containers on one Docker bridge network; clients run on the host (or on other hosts that can reach the gateway's published port). Replicas are assumed co-located on one low-latency segment: consensus is sensitive to round-trip time. Each replica exposes two ports: an HTTP port for the application API and reconciliation, and a separate port for the Raft transport.

- **Naming and discovery**

Clients know exactly one name: the gateway's address. The gateway is configured with the static list of replica HTTP URLs. Replicas are configured with their own Raft address and the Raft addresses of their peers. Within Docker, replica hostnames (`server-1`, `server-2`, `server-3`) resolve through the compose network's DNS; because a recreated container can reappear on a new IP, the consensus layer is configured to re-resolve peer names on every reconnect attempt rather than cache them. The gateway maps a replica's reported Raft leader address back to the matching HTTP URL by comparing hosts.

3.3 Modelling

- **Domain entities**

  The domain has one aggregate, the **Note**, one client-side record, the **Sidecar**, and three entities that stand for the system's own components, so that Note and Sidecar have somewhere to be mapped onto: **Client**, **Gateway** and **Node** (fig. 2).

  - **Note**: identified by its `name` (the primary key). It carries `content`, an integer `version` (incremented on every accepted write or deletion), a `content_hash` (SHA-256 of the content, used for corruption detection and for same-version divergence checks), an `updated_at` timestamp, and a boolean `deleted` tombstone flag.
  - **Sidecar** (client side): for a locally-downloaded note, a hidden JSON file (named `.` + the note name + `.syncnotes.json`) recording `{name, version}`: the base version for the next merge. It is bookkeeping, not authoritative, and may be deleted when the note is deleted.
  - **Client**: identified by `user`, the label shown in the corner of the screen (the OS username, unless overridden). Holds a local `sync_dir` and, for whichever notes have been opened in this session, their provisional Note copies and Sidecars.
  - **Gateway**: has no identity of its own, it is stateless. Its only attributes are the configured `replicas` list and the `leader` it last discovered among them.
  - **Node**, also called *replica*: identified by `replica_id`. Carries its consensus `state` (follower, candidate or leader), whether it currently `has_quorum`, and `peers_connected` out of `peers_total`, the same fields its own `/health` endpoint reports.

- **Mapping entities to components**

  Client, Gateway and Node correspond one to one to the three infrastructural components. Every **Node** holds the authoritative **Note** identically, because the cluster is a replicated state machine: there is no sharding, every node holds every note. The **Sidecar**, by contrast, lives only inside a **Client**, one file per locally-downloaded note, next to the note's own content, and nowhere on the cluster: it is not replicated, not authoritative, and the server code has no notion of it at all. A **Client** therefore holds only provisional **Note** copies and **Sidecars**, never

authoritative state. A **Gateway** holds no domain state at all: only a transient in-memory snapshot of the last note list it broadcast, purely to suppress duplicate notifications.

- **Domain events**
  - **Note**: *created*; *edited* (further classified on the client side as *saved as-is*, *merged with server changes*, or *conflict: your line was dropped*); *deleted*; *resurrected* (a write onto a tombstone).
  - **Sidecar**: *written* (a note is fetched or saved, recording the version it was based on); *cleared* (the note is deleted); *ignored as absent*.
  - **Node**: *leadership changed* (follower becomes candidate, candidate becomes leader, or a leader steps down); *quorum lost* / *quorum regained*; *peer connected* / *peer disconnected*; *fell behind* / *was repaired*.
  - **Client**: *note opened for editing* / *editing finished* (saved, or discarded); *live updates connected* / *live updates paused* (the background watcher's events stream drops and is retried with backoff); *local copy refreshed automatically*; *local copy removed* (the note was deleted elsewhere while it was not open for editing).
  - **Gateway**: *client subscribed to live updates* / *client unsubscribed*; *note list broadcast*; *no leader available* (every attempt to reach a leader failed for this request).
- **Messages exchanged**

  Queries: list notes, download note, fetch manifest (name → version/hash map, including tombstones), health. *Commands*: upload note (content, base version, op id), delete note (op id), trigger reconcile. *Events*: the note-list-changed server-sent event stream from gateway to clients. Internally, Raft *append-entries* / *request-vote* messages carry the replicated operations and elections.
- **System state**

  The authoritative state is the set of Notes with their versions, hashes and tombstones. Supporting that state, so that a later upload can merge against the right common ancestor and a retried write is not applied twice, requires a per-note version history and an idempotency ledger; those are a persistence-level concern rather than first-class domain entities. The consensus state (current term, log, leader) is managed by the Raft layer and persisted in a journal and periodic full-state dump per replica.

3.4 Interaction

- **Client and gateway: request/response**

  Every client operation is a single synchronous HTTP request to the gateway. The gateway never serves it locally: it discovers the current leader (by asking replicas for

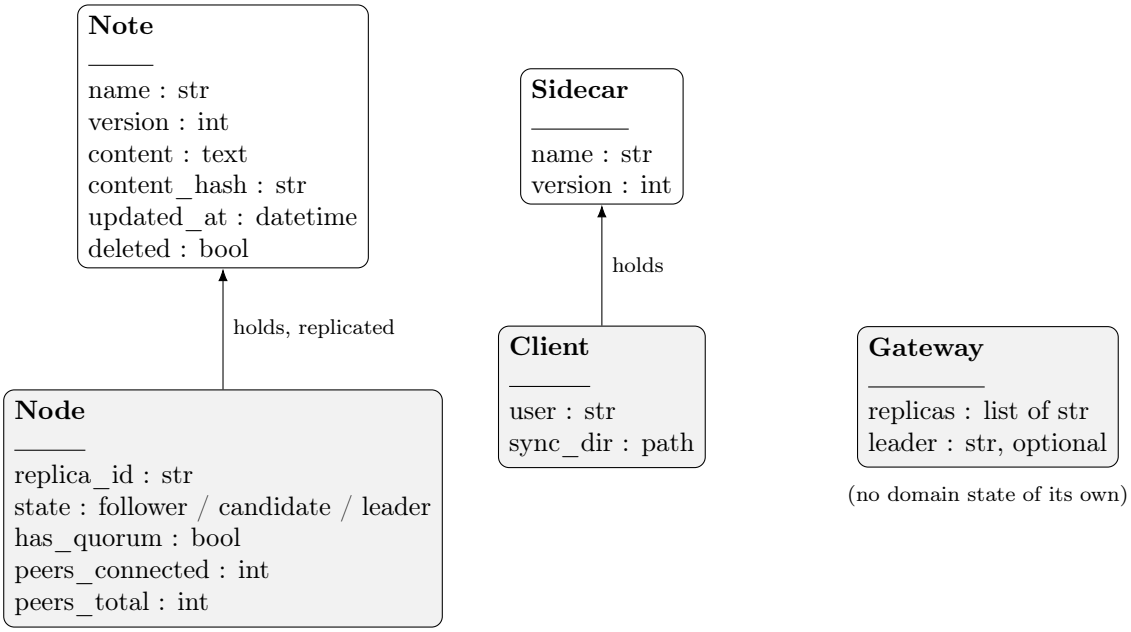

Figure 2: Data model. **Client**, **Gateway** and **Node** stand for the system's own components (section 3.2); the arrows show what each holds. **Note** is the single authoritative aggregate, held identically by every **Node**. The client-side **Sidecar** records which version of a note a **Client**'s local copy was last downloaded from, the base version used for the next three-way merge; it is bookkeeping, not authoritative. A **Client** also keeps a provisional copy of a Note's own content next to its Sidecar, not shown here as a separate box since it has no structure of its own beyond that text. A **Gateway** holds no domain state of its own. The version history and idempotency ledger that support the merge are a persistence-level concern, detailed in section 3.6.

their health until one reports a leader, then mapping that leader back to a known URL) and forwards the request there. If the leader is unknown or unreachable, or replies that it is no longer the leader, the gateway re-discovers and retries once more before returning a *no leader available* error. This realises N1 (the client is oblivious) and contributes to N4 (a failover is a transparent retry).

- **Gateway and clients: publish/subscribe for live updates**

  While the TUI is open, a background thread (separate from the thread drawing the screen) opens a long-lived *server-sent events* stream on the gateway and never returns from it. The gateway pushes the full note list whenever it changes. Two things can change it: an upload/deletion that went through this gateway (pushed immediately), and an out-of-band change such as reconciliation catching a replica up (caught by a low-frequency poll the gateway runs only while at least one client is subscribed). A periodic comment line keeps the stream alive through idle periods. This realises F6.

- **The editing workflow**

  Figure 3 shows a save. The client downloads a note (recording the base version in the sidecar), the user edits locally, and the client uploads `(content, base_version, op_id)`. The gateway forwards to the leader. The leader submits the write to the Raft log; once the entry is committed by a quorum, *every* replica applies it by running the same merge function over its own copy. The leader returns the merged content and a flag saying whether any of the client's lines were dropped. The client overwrites its local copy with the returned content and updates the sidecar to the new version.

- **Replica and replica: consensus and reconciliation**

  Replicas continuously exchange Raft messages (heartbeats, log replication, elections). Independently, every non-leader replica periodically pulls the leader's manifest and fetches any note it is missing, behind on, or diverging on at equal version, and additionally re-hashes its own stored contents to catch local corruption. This pull-based catch-up is a form of anti-entropy [Demers et al., 1987], layered on top of consensus rather than replacing it. Reconciliation is one-directional (follower pulls from leader) and best-effort; it is a safety net, not the primary replication mechanism.

3.5 Behaviour

- **Client, the TUI**

  The client runs two threads. The main thread is a curses event loop: it redraws the note list and waits for a key, but never blocks indefinitely, a fixed timeout wakes it every `REFRESH_TICK_MS` so it also notices changes pushed by the other thread even while the user presses nothing; this non-blocking loop is what keeps the screen

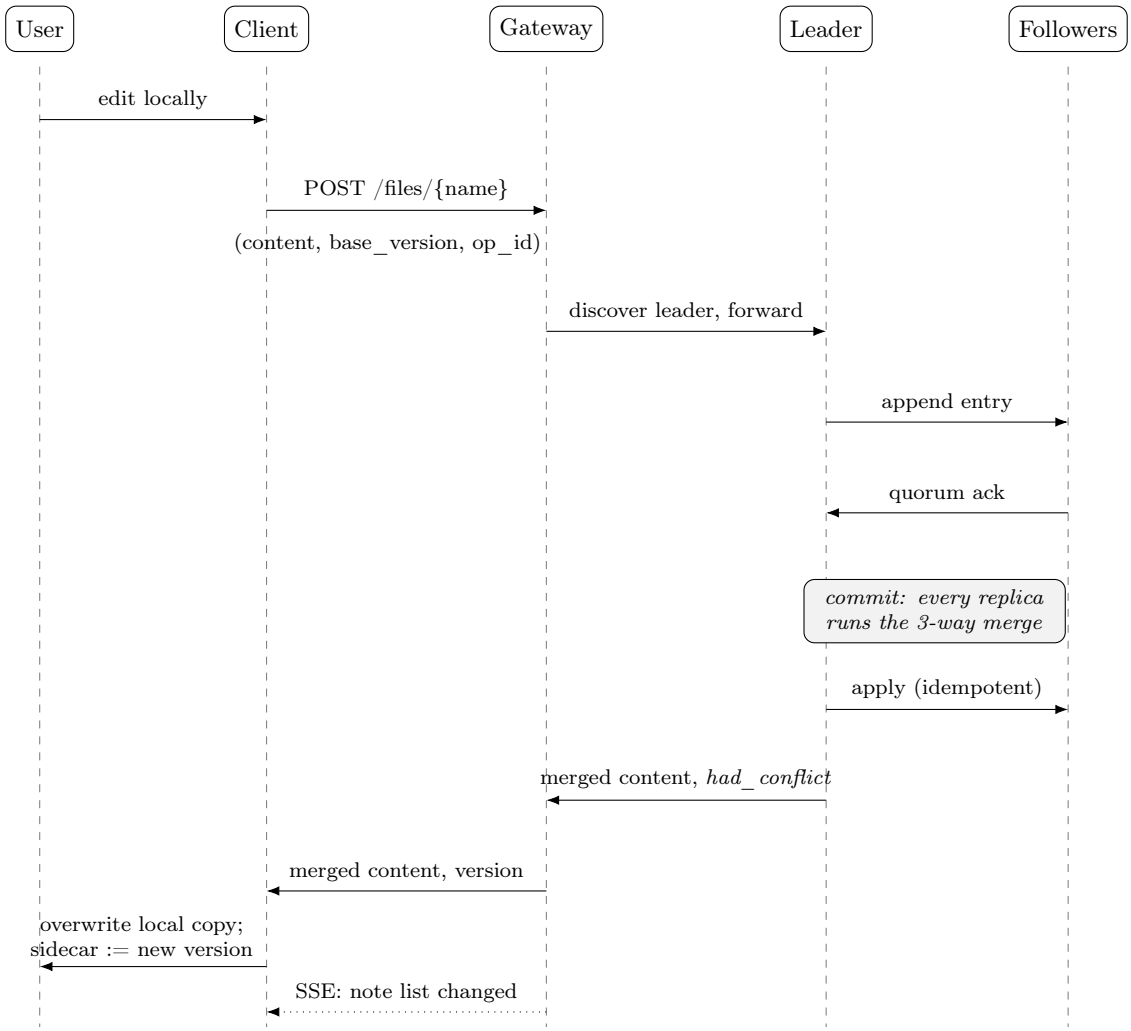

Figure 3: Saving a note. The merge runs during Raft *commit*, on every replica, so its result depends only on the log order, not on which replica evaluated it.

responsive no matter how the cluster is behaving (N10). The second thread is the background watcher: it holds the gateway's events stream open and, for every note list it receives, refreshes any locally-cached note that is now stale and drops the local copy of anything that was deleted, all without the user asking. The two threads only meet at one small piece of shared state, the current note list, the last status message and which note (if any) is open in the editor right now, all behind one lock. That last field is a deliberate behavioural rule: while a note is open for editing, the watcher must not touch that note's local file or its sidecar. Without it, a background refresh could rewrite the sidecar's base version between the moment the user started editing and the moment they save, and the upload would then declare a base version the in-buffer edit was never actually taken from, corrupting the three-way merge (F3) on data that looked perfectly fine. The client is otherwise stateless towards the cluster: it never decides what the authoritative content is, it only decides which of its own local copies to keep, refresh or edit.

- **Gateway**

  Holds no durable state. For each request it performs leader discovery and forwarding with one retry. For the events stream it maintains only the set of current subscribers, a single background poll task (alive only while there are subscribers), and the last list it broadcast. It never mutates cluster state.

- **Replica: note store**

  The note-store operations *upload* and *delete* are the replicated operations. When Raft commits one, the replica executes it against its local database:

  - *Upload.* If the operation id is already in the ledger *and* the note is currently live, return the stored result unchanged (duplicate, N9). Otherwise: look up the base content (the archived version named by `base_version`, if any); compute the merge of (base, current authoritative, uploaded); write the merged content, bump the version, archive the new version, clear any tombstone, and record the operation id. Uploading a name with no existing server content is exactly how a note gets *created* (F4). A write onto a tombstoned note is treated as a fresh note (the dead content is not merged back in): this is the "resurrect" path.
  - *Delete.* (F5) If the note is already tombstoned, no-op (no second version bump, N9). Otherwise bump the version, set the tombstone, archive the (unchanged) content under the new version, and record the operation id. If the note was never seen, still write a tombstone row, so a later stale reconciliation cannot resurrect it, even for a note that went missing while this replica was offline (N8).

- **The merge rule**

  Given `base` (common ancestor, possibly absent), `server` (current authoritative, possibly absent) and `client` (uploaded):

1. If there is no server content, accept the client content verbatim (new note).
2. If there is no base, keep the server content unchanged: the client had nothing to diff against, so none of its edit can be trusted. The user is told to download again before editing.
3. If base equals server (server untouched since the download), accept the client content.
4. If base equals client (client did not actually edit), keep the server content.
5. Otherwise, compute line-level change hunks of server-vs-base and client-vs-base using a longest-matching-block sequence comparison. Apply *all* server hunks. Apply only those client hunks that do not overlap any server hunk. If any client hunk was dropped for overlap, set `had_conflict`. At a shared boundary the server hunk sorts first.

Rule 5 is "the server always wins on conflicting lines" made precise. `had_conflict` exists so the client can distinguish a clean three-way merge from one where some of the user's lines were silently overwritten; both otherwise look like "the content changed".

- **Replica: consensus role**

Each replica is a follower, candidate or leader (fig. 4). Followers that miss the leader's heartbeats for an election timeout become candidates and call an election; a candidate with votes from a quorum becomes leader. Only a leader accepts client writes and appends to the log. A leader that cannot reach a quorum steps down. Every state transition is logged, which is what makes an election or a rejoin visible in `docker compose logs`.

- **Replica: reconciliation role (background)**

On a fixed interval, a non-leader replica runs one reconciliation pass against the leader (fig. 5). A leader's reconciliation task idles (it has nobody authoritative to pull from). Reconciliation is also exposed as an on-demand endpoint, used by the corruption chaos script.

- **Which component updates the state**

Only the cluster, and within it only through a committed Raft operation (for normal writes) or through `apply_if_newer` during reconciliation (for catch-up and repair). The gateway and client never write authoritative state.

3.6 Data and Consistency Issues

- **What is stored, where and why**

The notes and their history must be stored because the cluster is the sole source of truth and because editing needs a durable common ancestor for merging. Storage is

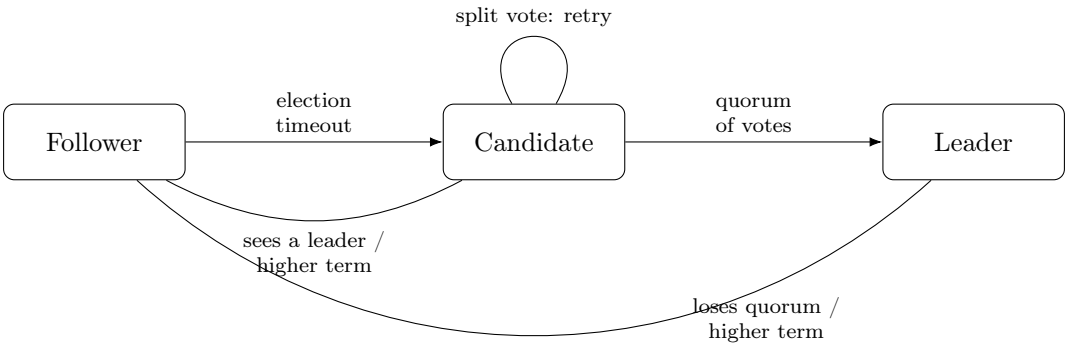

Figure 4: Consensus role state machine per replica. Client writes are accepted only in the Leader state and only while a quorum is reachable.

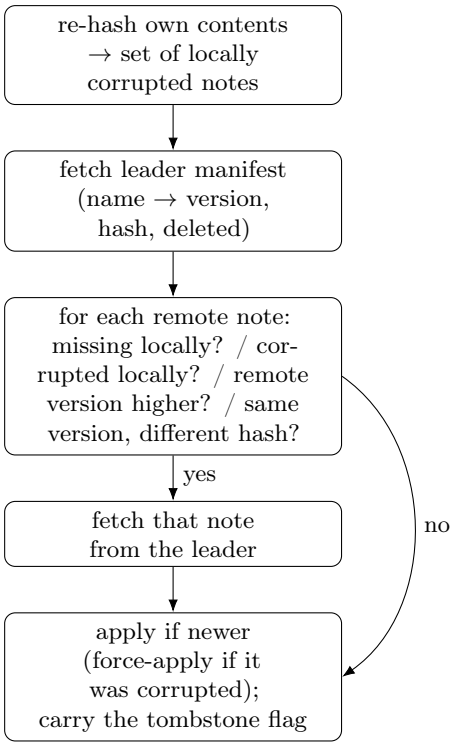

Figure 5: One reconciliation pass (non-leader only). Step 1 catches corruption that consensus cannot see, because a storage-level overwrite is not a log event.

per replica: each node has a full, independent copy. This satisfies N2's convergence clause literally (three databases that end up identical) and makes I2's "one SQL file per replica" demonstrable.

- **How persistent data is stored**

  As relational tables: `files` (one row per note), `file_versions` (one row per (note, version)), `applied_operations` (one row per applied write). Relations are the natural fit, a fixed narrow schema, primary-key lookups by name and by (name, version), no nesting, and a SQL engine gives transactions for free, which the apply path relies on. The Raft log itself is persisted separately by the consensus layer (a journal plus periodic full-state dumps) so a restarted replica can rebuild its committed state.

- **Who queries the database, when and why**

  The leader's HTTP handlers read `files` on every list/download (F1, F2) and read `file_versions` during a merge. Every replica writes `files`, `file_versions` and `applied_operations` when a Raft operation commits. A non-leader replica reads its own tables (manifest, self-hash) and writes them during reconciliation. Concurrent *reads* occur (several client requests, the events poll); each uses its own short-lived session. Concurrent *writes* to the same note do not occur at the SQL level, because Raft serialises the operations that produce them: this is how N3 is enforced without database-level locking.

- **Data shared between components**

  Only the authoritative note state, and only within the cluster; it is shared by *full replication*, not by a shared datastore. The client and gateway hold no shared state. The sidecar is private to one client.

- **Consistency model**

  Writes are **linearizable** with respect to the Raft commit order: each upload/delete takes effect atomically at a single point, and the merge outcome is a deterministic function of the prefix of the log before it. Reads go to the leader only, giving clients **read-your-writes** and **monotonic reads**. A follower may transiently lag, but a follower is never read by a client; reconciliation bounds how long a follower can stay behind. There is no support for reading from followers precisely because best-effort reconciliation does not, on its own, guarantee they are caught up.

- **Known consistency subtleties**

  An upload with an *unknown* base version against an existing note keeps the server content entirely and discards the client's, conservative by design (behaviour rule 2).

  Recreating a deleted name comes back at `tombstone_version + 1`, not at 1, because version is monotone per name.

The idempotency ledger is honoured only while a note is live: a create and a later recreate of the same empty note hash to the same operation id, so once the note is tombstoned that stale match must not block the recreate.

3.7 Fault-Tolerance

- **Replication**

  The authoritative state is fully replicated on all three nodes through the Raft log: a write is committed only after a quorum has durably appended it, so the loss of any one node loses no committed data (N4). This is *active* replication (every replica executes every operation), which is what lets the merge run identically everywhere (N3).

- **Reconciliation as second-line replication**

  Because a follower could still miss an apply (crash between commit and local write, or a message lost during a partition), a follower also *pulls*. Each pass diffs its manifest against the leader's and fetches anything missing, older, or hash-diverged at equal version, then force-repairs anything its own self-hash flagged as corrupt (N6). This closes gaps that consensus alone would leave and, crucially, catches corruption that never went through the log.

- **Heartbeating, timeouts, retries**

  Raft heartbeats from leader to followers detect a dead leader within an election timeout and trigger a new election (N4). The gateway applies a short HTTP timeout to every replica call and treats a timeout like an unreachable leader, re-discovering and retrying once. The client applies its own request timeouts and, in the UI's background watcher, an exponential reconnect backoff on the events stream. The reconciliation loop runs on a fixed interval regardless of outcome.

- **Address changes**

  A recreated container returns on a new IP. The consensus layer is configured to *not* cache DNS resolutions (and to cache resolution *failures* only briefly), so surviving peers re-resolve the returning node's name and re-admit it within seconds instead of dialing the dead address for the default cache lifetime (N7).

- **Error handling**

  A replication failure (no leader, queue full, leader changed mid-flight, timeout) is surfaced to the gateway as a 503 with a reason; the gateway retries once, then returns 503 to the client, which shows "the cluster has no leader right now, retry in a moment". This is the same path taken when quorum is lost entirely, not just partitioned: a lone surviving replica never gets enough votes to commit a write, so the request eventually times out and gets this same refusal instead of that replica answering on its own authority (N11). A missing note is a 404. A corrupt sidecar on the client is treated as no sidecar (safe: the next upload becomes conservative). A

reconciliation HTTP error is logged and retried next interval. A malformed request body is rejected by request validation before reaching any logic.

- **What happens when each component fails**

  A follower crashes: quorum is still 2/3, service continues; on restart it rejoins and reconciles. *The leader crashes*: a follower is elected, the gateway re-routes, service resumes; the old leader rejoins as a follower. *The gateway crashes*: clients cannot reach the cluster until it restarts (it is stateless, so restart is clean); this is the one single point of failure. *A client crashes*: no cluster impact; its provisional edits are simply lost.

3.8 Availability

- **Caching**

  None on the server side: every read hits the leader's database, which is local and cheap. The gateway keeps only a transient snapshot of the last broadcast note list, purely to avoid pushing identical updates; it is not a read cache and is never served to a direct request. The client keeps local note copies, but always re-reads from the cluster before an edit.

- **Load balancing**

  Not applicable in the traditional sense: all reads and writes must go to the single leader for consistency, so there is nothing to balance across replicas. The gateway distributes *nothing*; it routes everything to one place.

- **Behaviour under network partitioning**

  Consistent with the CP choice (N5):

  - The side with a quorum (majority) keeps a leader and keeps serving reads and writes normally.
  - A replica in the minority loses quorum: it cannot be leader, cannot commit writes and reports `has_quorum = false` on its health endpoint. It does not serve stale data as authoritative because the gateway only ever talks to a replica that reports itself leader with quorum.
  - There is therefore no split brain and no divergence: at most one side makes progress.
  - When the partition heals, the minority replica catches up through Raft log replication and through reconciliation, converging to the majority's state.

  The accepted cost is that clients whose gateway can only see the minority get temporary errors until the partition heals; availability is sacrificed on that side, deliberately.

3.9 Security

The current design assumes a **trusted network**: the cluster and gateway run on a private segment and clients are operated by trusted team members.

- **Authentication**

  No endpoint authenticates the caller. In a hardened deployment the gateway would be the natural place to terminate client authentication (an API token or mutual TLS), and the replica HTTP ports (application API, manifest, reconcile) would be restricted to the cluster network.

- **Authorization**

  Every user may read and write every note, and there is a single role. Note-level ownership or an ACL model could be added later without changing the replication design.

- **Cryptography**

  None in transit (plain HTTP and the plain Raft transport). At rest, the SHA-256 content hash is used for *integrity* checking (corruption detection), not for confidentiality or authenticity.

4 Implementation

- **Network protocols**

  HTTP everywhere on the application-facing side: client to gateway, gateway to replica, and replica to replica for reconciliation. The consensus layer is the one exception: Raft peer traffic runs over its own plain TCP connection on a separate port, not over HTTP.

- **In-transit data representation**

  JSON on every HTTP surface, generated and parsed through Pydantic models. The Raft transport carries the consensus library's own internal wire format instead: `pysyncobj` serialises each message with Python's `pickle` and compresses it with `zlib` before writing it to the socket; this is entirely internal to the library and not something SyncNotes' own code touches.

- **Database queries**

  SQL, through the SQLAlchemy ORM (I2). Every query is a primary-key lookup.

- **Authentication and authorization**

  Not applicable: no technology was chosen because none is used, by design.

4.1 Server cluster

- **Language and runtime**

  Python 3.12 (I1), the version pinned by the container's `python:3.12-slim` base image. Each replica is a single process.

- **Web framework and server**

  The application API is a FastAPI. It gives declarative request/response models with automatic validation and JSON (de)serialisation, which covers the *malformed request rejected before any logic* point for free. The app's lifespan hook creates the consensus object and starts the reconciliation and cluster-heartbeat background tasks on startup and tears them down on shutdown.

- **Consensus**

  Raft. The note store is a subclass of the consensus layer's replicated-object base; the two write operations are plain methods wrapped with a replication decorator, so calling one submits it to the Raft log and it executes on every replica when committed. Configuration disables dynamic membership changes (the cluster is fixed at three), sets the DNS resolution cache to zero and the failure cache to five seconds (N7), points the journal and full-dump files at the data volume, and registers a state-change callback that logs every Follower/Candidate/Leader transition.

  An `async` adapter bridges the consensus layer's callback-style API to FastAPI: the HTTP handler submits the replicated call with a callback that resolves an `asyncio` future and awaits that future with a 10-second timeout. A timeout is reported as a synthetic failure reason distinct from the consensus layer's own (queue full, missing leader, not leader, leader changed, ...), and every non-success reason becomes a 503.

- **Persistence**

  SQLite. Three mapped classes (`FileRecord`, `FileVersion`, `AppliedOperation`) map to the three tables of section 3.6. `check_same_thread=False` is set because the consensus tick thread and the Uvicorn worker thread both touch the engine; each unit of work opens and closes its own `Session`. A tiny hand-rolled migration adds the `deleted` column in place if an older database predates it.

- **Reconciliation**

  An `asyncio` task loop. Each pass uses an `httpx` async client (5-second timeout) to GET the leader's `/internal/manifest` and, for each note it needs, GET `/files/{name}` (which returns tombstones too, with `deleted=true`). Local corruption is found by re-computing the SHA-256 of every stored content and comparing with the stored hash. `apply_if_newer` performs the conditional write (apply if forced, or missing, or strictly newer, or equal-version-but-different-hash).

HTTP API

Endpoint	Purpose
`GET /health`	Consensus role, current leader, quorum flag, connected/total peers. Used by the gateway for discovery and by the chaos scripts.
`GET /files`	User-facing note list; tombstones hidden; sorted by name.
`GET /files/{name}`	Full note including content and hash; returns a tombstone with `deleted=true` (the gateway turns that into 404).
`POST /files/{name}`	Upload: body is `{content, base_version, op_id}`; goes through the replicated log; returns merged content, new version and `had_conflict`.
`DELETE /files/{name}`	Tombstone the note through the replicated log; idempotent.
`GET /internal/manifest`	name → {version, hash, deleted} for every row, tombstones included. (reconciliation)
`POST /internal/reconcile`	Run one reconciliation pass now; returns the list of repaired notes.

- **Idempotency keys**

  The client derives the upload operation id deterministically as `sha256(name \0 base_version \0 content)`, so an identical retry after a 503 carries the same id and the ledger dedups it, with no client-side state to persist across the retry. The delete key folds in the note's current version, so a retry of the same delete dedups while deleting a *recreated* name gets a fresh key.

4.2 Gateway

Python 3.12 (same container pin as the server), FastAPI + Uvicorn, `httpx` for outbound calls, no database. It is configured with the list of replica URLs. Leader discovery asks each replica's `/health` until one names a leader, then matches that Raft host against the configured URLs. Forwarding retries once across re-discovery; a 503 from the leader is treated as *stale, try again*, any other 4xx/5xx is passed through. The events endpoint is a `StreamingResponse` of `text/event-stream`; the poll loop, the subscriber set and the last snapshot are module-level state guarded by an `asyncio` lock so that a subscriber (re)connecting exactly as the loop tears down is not left without a poller.

Endpoint	Purpose
`GET /health`	Configured replicas and the leader currently discovered among them.

Endpoint	Purpose
`GET /files`	Forwards to the leader's `/files`.
`GET /files/{name}`	Forwards to the leader; a tombstone in the response becomes a 404.
`POST /files/{name}`	Forwards the upload; broadcasts the new note list to subscribers on success.
`DELETE /files/{name}`	Forwards the deletion; broadcasts on success.
`GET /events`	Server-sent-events stream of the note list, pushed on change and polled in the background while subscribed.

4.3 Client

Python 3.12+

- `tui.py`

  The curses full-screen UI and the actual F7 entry point, opened directly by `syncnotes.py` at the repo root. The main loop redraws on a short input timeout so background updates surface without a keypress, and dispatches a single keystroke at a time: arrows/`j`/ `k` to move the selection, `Enter` to open the note in a curses `Textbox` editor, `n` to create one (name validated, guarded against clobbering an existing note), `d` to delete (with a confirmation prompt), `r` to force-refresh, `q`/Esc to quit. A daemon thread consumes the gateway's event stream with exponential back-off on disconnect and pushes note-list and status updates through a lock-protected hand-off object (`_LiveNotes`), which also tracks which note is currently open for editing so the watcher never touches its file mid-edit.

- `api.py`

  Thin HTTP calls (list, download, upload, delete, watch events) and the two idempotency-key functions.

- `sync_state.py`

  Read/write/clear the hidden JSON sidecar.

- `notes.py`

  The workflow the TUI is built on: fetch, open (download + write local + sidecar), save (read sidecar → upload → write back merged content + sidecar), delete, create (guarded against clobbering an existing name), and human wording of every outcome. All network errors are translated here into user-facing sentences.

- `cli.py`:

  `list`/`download`/`upload`/ `delete` subcommands kept for scripting (the chaos scripts and `demo_collaborative_edit.py` drive simulated clients through it), plus a `tui` subcommand that opens `tui.py`.

- **TUI concurrency**

  The curses main loop uses a short input timeout so it keeps redrawing and picking up background updates even with no keypress. A daemon watcher thread consumes the gateway's events stream, refreshes stale local copies and removes local files for notes deleted elsewhere. A small lock-protected hand-off object (`_LiveNotes`) carries the current note list, a pending status line and *which note is open for editing*: the watcher must not touch a note's local file or sidecar while it is being edited, or the base version read at save time would no longer match the buffer. The check-and-write is done atomically under the same lock the "begin editing"/"end editing" calls use.

- **Input safety**

  The editor refuses to open a note that does not fit the window (curses' Textbox only captures what is on screen, so opening a larger note would truncate it on save). New-note names are validated against: empty/whitespace, leading/trailing spaces, `/` or `\`, a leading dot, Windows-reserved characters and control characters.

4.4 Technological details

Concern	**Choice**	**Why**
Consensus	pysyncobj 0.3.15 (Raft)	Understandable protocol; `@replicated` keeps the footprint tiny; pure Python, no external service.
Persistence	SQLite + SQLAlchemy	Embedded, zero-config; one file per replica makes independent convergence visible (I2).
Web API	FastAPI + Uvicorn	Declarative validation, JSON models, async lifespan for background tasks.
Client HTTP	`requests`	Synchronous, simple; the client has no concurrency need beyond one watcher thread.
Reconcile / gateway HTTP	`httpx` (async)	Non-blocking calls inside the `asyncio` loops.
Live updates	Server-Sent Events	One-way server→client push over plain HTTP; trivial to consume with `requests` streaming (F6).
Merge	Standard-library `difflib`	`SequenceMatcher` line-level opcodes with `autojunk` disabled give stable hunks. (Ratcliff/Obershelp matching, not Myers: the report describes the mechanism, not the library's internals.)

Concern	Choice	Why
Packaging	Docker + compose	Containerised deployment; `python:3.12-slim` base images.

5 Validation

Validation has four layers: automated unit tests per component, automated integration tests across component boundaries, an end-to-end acceptance demo of the ordinary collaborative-editing use case against a real cluster, and scripted chaos scenarios that exercise the distributed failure modes end to end. The automated suite has 167 tests, all passing.

5.1 Automatic Testing

- **How to run**

```
python3 -m venv venv
source venv/bin/activate
pip install -r server/requirements-dev.txt -r client/requirements-dev.txt
cd server  && python -m pytest
cd ../gateway && python -m pytest
cd ../client  && python -m pytest
```

- **Strategy** Each source module has a dedicated test module. Infrastructure at the edges (outbound HTTP, curses, the filesystem where it matters) is either mocked or pointed at a temporary location, so most tests exercise business logic without a running cluster. Tests assert observable behaviour (returned values, resulting rows, logged warnings), not internal call counts.

  - **Server: note store and merge (unit)** — `server/tests/`

    `test_merge.py` covers every branch of the merge rule: a new note takes the upload whole; an unknown base keeps the server content; server-untouched accepts the client edit; client-unedited keeps the server content; disjoint edits both survive; a same-line conflict keeps the server version; a server deletion of a line beats a client edit to it; a client deletion of a server-untouched line is honoured; a client append survives next to an unrelated server edit. *(F3, N3.)* `test_db.py` covers version monotonicity, the "apply if newer / same-version different-hash / forced" matrix, tombstone creation and idempotent re-deletion, resurrection of a tombstone by a new write, recreation after deletion even with a colliding operation id, the manifest-includes-tombstones / summary-hides-them split and self-hash corruption detection. *(F4, F5, N6, N8, N9.)*

- **Server: HTTP surface (integration)** — `server/tests/`

  `test_server.py` drives the FastAPI app with a test client over an in-memory database: health shape, 404 on missing/deleted, upload/download round-trip, version increment, op-id dedup, list and manifest contents, conservative behaviour without a base version, clean apply with the correct base version, the two-client disjoint vs. same-line scenarios, delete hiding the note, double-delete not bumping again, delete-then-recreate, deleting an unknown note and the conflict warning log. *(F1-F5, N2, N3, N9.)*

- **Server: reconciliation (integration)** — `server/tests/`

  `test_reconciliation.py` (async) covers pulling a missing file, pulling a stale file, repairing a same-version hash divergence, repairing local bit rot even when the hash column still matches the peer, applying a deletion the peer made, *not* resurrecting a tombstone from a stale manifest and being a no-op when already up to date. *(N6, N8.)* `test_cluster.py` covers the leader-URL mapping (none when no leader, none when self is leader, host-to-HTTP-port rewrite otherwise). `test_quorum.py` is the one exception to "no running cluster": it starts three real `NotesStore` (Raft) nodes in-process, over local sockets, with no HTTP layer and no Docker. It asserts that killing the minimum number of replicas needed to break quorum drops `has_quorum` on the survivor, that a write submitted in that state never commits, and that quorum returns once the stopped replicas restart. *(N11.)*

- **Gateway (integration)** — `gateway/tests/`

  `test_gateway.py` covers leader discovery, routing of every verb to the leader, turning a tombstone download into a 404, 503 when no leader is discoverable, re-routing after a stale leader rejects a write, re-routing when the leader is unreachable, and the events stream (initial snapshot on connect, broadcast on a polled change, immediate broadcast on upload, heartbeat when idle, subscriber bookkeeping across reconnect). *(N1, N4, F6.)*

- **Client: CLI dispatch (unit)** — `client/tests/`

  `test_cli.py` covers the four scriptable subcommands: the download/upload round-trip with the sidecar, sending no base version absent a prior download, reporting a server-side merge or a dropped edit, and the delete-with-local-cleanup path. The `tui` subcommand carries no logic of its own beyond dispatch, so it is not separately tested here.

- **Client (unit)** — `client/tests/`

  `test_api.py`, `test_notes.py`, `test_sync_state.py`, `test_tui.py` cover URL construction, the deterministic op-id (stable across an identical retry, version-sensitive delete key), sidecar round-tripping, the open/save/delete/create workflows and their friendly error wording, the base-version-from-last-open behaviour, writing back server-merged content, the outcome-wording matrix (clean / merged / conflict / full discard), and

the curses-free TUI logic (name validation, window-fit check, edited-text normalisation, the editing lock in `_LiveNotes`, the scrolling list offset (stays put while the selection is visible, follows it past either edge, clamps when the list shrinks below the old offset), and every branch of "sync local copies": skip never-downloaded, skip currently-editing, skip already-latest, re-download stale, best-effort on failure, remove a deleted note's local copy, do not wipe on an empty list, do not write if editing started during the download). *(F2, F3, F6, F7, N10.)*

5.2 End-to-end merge demo

`scripts/demo_collaborative_edit.py` is the one script that is not a chaos scenario: it injects no fault, it demonstrates the ordinary use case end to end. It drives the real client upload/download workflow from four separate "user" directories against a real cluster through both merge scenarios: two clients editing disjoint lines (both survive) and two clients editing the same line (the server's version wins and the losing client is told so), asserting on the actual merged content returned by the gateway. This is the acceptance evidence for F3 that `test_merge.py` and `test_server.py` cannot provide on their own, since both stop short of a real multi-process cluster.

5.3 Chaos scenarios

Each script, living under `scripts/` as `chaos_⟨name⟩.py`, brings up a clean cluster with `docker compose up --build`, injects one fault, asserts recovery with `[OK]`/`[FAIL]` lines, tears down with `docker compose down -v`, and exits non-zero on the first failure.

Script	What it injects and asserts	Req.
`kill_leader`	Writes through the gateway, `docker kill`s the current leader, waits for a new leader among the survivors, writes again through the gateway (no reconfiguration), asserts the version advanced by one and a read-after-write is consistent.	N4, N1
`network_partition`	`docker network disconnect`s one follower. Asserts the majority keeps serving, the isolated replica reports `has_quorum=false` (no split brain), and on reconnect it converges to the latest content.	N5
`corruption`	Overwrites a follower's stored content directly in SQLite, leaving the hash column intact (simulated bit rot, invisible to Raft). Triggers reconciliation and asserts the correct content is restored from the leader.	N6

Script	What it injects and asserts	Req.
`restart_node`	`docker compose up -d -force-recreate`s a follower so it returns as a new container on a new IP, while writes continue through the majority. Asserts the peers re-admit it and it catches up every missed write.	N7
`delete_converges`	Stops a follower, deletes a note through the gateway while it is down, restarts the follower. Asserts the tombstone is present on all three replicas' own databases and the note stays absent from the list: a physical row delete would pass "delete then GET 404" yet let the stale follower re-add the note.	N8

- **Supporting tooling**

  `scripts/inspect_db.py` prints each replica's own SQLite rows straight from the file, bypassing the HTTP API, used in a demo to show the cluster really is three independent stores that converge (I2's acceptance criterion); `--watch` re-dumps every 1.5s for a live view.

5.4 Acceptance test (manual)

A few scenarios are validated by hand, during development, because automating them well is disproportionate to the payoff. The procedure for each:

- **TUI feel**

  Open two real terminal sessions on the same note and check: the list updates live in one when the other saves; a non-edited note's local copy is refreshed automatically; a same-line edit produces the "your line was dropped" status; and resizing the terminal or stopping the cluster never freezes the UI (N10). Not automated: curses behaviour under a real terminal is awkward to assert meaningfully.

- **Multi-client convergence**

  Run several clients writing in a loop and watch `inspect_db.py` to see the three databases track each other. Not automated: it is an observational check, and the chaos scripts already assert the hard invariants.

- **Full cluster cold start**

  Run `docker compose up --build` from scratch on a clean machine and check that the three replicas form a quorum and the gateway answers on `:8080`.

6 Deployment

6.1 Prerequisites

Software	Version	Purpose
Docker Engine	20.10+	Runs the cluster and the gateway.
Docker Compose	v2	Brings the whole system up from one file.
Python	3.12+	Runs the client and the chaos scripts on the host.

6.2 Container images

Both images are built from `python:3.12-slim`. Each copies its `requirements.txt`, installs it, copies `src/`, sets `PYTHONPATH=/app/src`, and runs Uvicorn: `server.main:app` on port 8000 for a replica, `gateway.main:app` on port 8080 for the gateway.

6.3 Compose topology

`docker-compose.yml` defines four services on one bridge network `syncnotes` (fig. 6):

- `server-1`, `server-2`, `server-3`: identical image, differing only in environment: `REPLICA_ID`, `DB_PATH=/data/server.db`, `RAFT_SELF_ADDR=server-`i`:9000`, `RAFT_PEER_ADDRS` (the other two), and the journal/dump file paths. Each has its own named volume mounted at `/data`, so its database and Raft log survive a container restart.

- `gateway`: `REPLICA_URLS` listing the three replica HTTP URLs; publishes container port 8080 to host `8080`; `depends_on` the three replicas.

Only the gateway's port is published; the replica HTTP and Raft ports stay on the internal network.

6.4 Procedure

```
# Terminal 1 - repository root (where docker-compose.yml lives)
# Docker Desktop (or an equivalent daemon) must already be running here,
# otherwise the next line fails with "Cannot connect to the Docker daemon"
# no venv needed: the cluster and gateway run inside the containers

docker compose up --build -d        # build images, start detached
docker compose logs -f              # follow logs across restarts (Ctrl+C to
    stop following)
curl -s http://localhost:8080/health   # {"status":"ok", "leader": "..."}
```

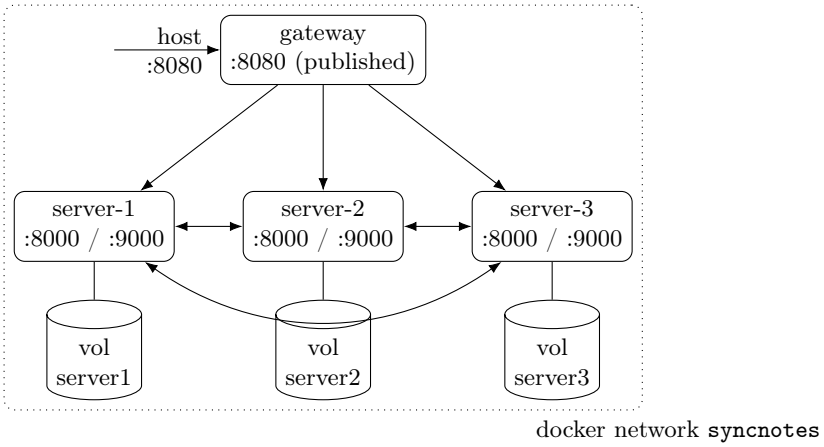

Figure 6: Compose deployment: three replica containers, each with a private named volume, plus the gateway, the only service with a published port.

Expected outcome: three replicas elect a leader within a few seconds (visible as `raft state:` and `cluster: ... leader=... quorum=True peers=2/2` log lines), and the gateway answers on `:8080`. The client (Client installation below) is started separately, in its own terminal, once this is up.

- **Operational note**

  Do not run `docker compose up` in the foreground if you intend to stop/start individual replicas: that foreground stream attaches once and a replica you restart disappears from it even though it rejoined. Use `-d` plus `docker compose logs -f`.

- **Configuration parameters**

  `RECONCILE_INTERVAL_SECONDS` (default 30): reconciliation period; `CLUSTER_HEARTBEAT_SECONDS` (default 15): how often the cluster summary line may be re-logged; `EVENTS_POLL_INTERVAL_SECONDS` (default 1) and `EVENTS_HEARTBEAT_SECONDS` (default 15) on the gateway.

6.5 Client installation

```
# Terminal 2 - repository root, once the cluster above answers on :8080
# separate venv from the server/gateway containers (own dependencies)

python3 -m venv venv
source venv/bin/activate
pip install -r client/requirements.txt
python3 syncnotes.py                # launches the full-screen note browser (
    F7)

# later launches: just re-activate and re-run, no need to reinstall
```

```
# source venv/bin/activate && python3 syncnotes.py
```

7 User Guide

The full-screen note browser is the system's only user-facing interface (F7). Run `python3 syncnotes.py` from the project root. The screen shows the team's shared notes.

Key	Action
↑ / ↓ (or `k` / `j`)	Move the selection
`Enter`	Open the selected note for editing
`n`	Create a new note (you are asked for a name)
`d`	Delete the selected note (a `y/n` confirmation is asked)
`r`	Refresh the list now
`q` / `Esc`	Quit

If there are more notes than fit on screen, the list scrolls automatically as the selection moves past either edge.

Inside the editor, type normally; `Ctrl+G` saves and exits, `Esc` discards. After a save, the status line reports the outcome: *saved as-is*, *server merged in changes since your last download*, or *someone else edited the same lines: your changes to those lines were dropped.*

- **What happens automatically**

  While the browser is open, the list updates when anyone changes a note and your local copies of notes you have opened are re-downloaded when they change elsewhere, except a note you are editing right now, which is left alone until you finish. If someone deletes a note you had opened, its local file disappears on the next update. Local copies live in `~/SyncNotes/` (override with `--dir`); point at a non-default gateway with `--gateway`; override the username shown in the corner with `--name`.

- **If the cluster is unreachable**

  The browser keeps working and shows "Cannot reach SyncNotes, is the server running?" or "The cluster has no leader right now, retry in a moment". It never freezes; `q` always quits.

8 Self-evaluation

8.1 Marzia De Maina

As the sole author I made every design decision and wrote the application code and the deployment myself; the automated test suite was AI-generated and then reviewed by me. The areas I judge strongest are the replicated-merge design and the failure-mode

validation through the chaos scripts; the areas where I had to cut scope, and know it, are security and the gateway's availability.

- **Strengths of the result**
  - *Coherent consistency story*: the CP choice is followed through from the requirements to the code: writes are linearised by a Raft log, the merge runs inside the replicated operation so its outcome depends only on log order, and a minority partition is made to stop rather than diverge.
  - *Defence in depth for integrity*: consensus handles the normal case; reconciliation plus a per-note content self-hash handles the case consensus is structurally blind to (storage corruption that is not a log event). The deletion design (tombstones through the same replicated path, never a physical row delete) was specifically built so a note cannot be resurrected by a replica that missed the delete.
  - *Genuinely usable by a non-technical person*: the full-screen UI needs no commands or paths, validates input with specific messages, keeps local copies in sync on its own, and never blocks on a slow or dead cluster.
  - *Correct failure visibility*: the one case the system does *not* hide (total loss of quorum) is surfaced as a plain-language temporary error rather than pretended away, which is the correct behaviour for a CP system.
- **Weaknesses of the result**
  - *The gateway is a single point of failure:* it is stateless and restarts cleanly, but while it is down no client can reach the cluster. A second gateway instance behind a virtual IP, or teaching the client the replica list as a fallback, would remove this.
  - *No security whatsoever*: no authentication, authorization or transport encryption; the trusted-network assumption is a real limitation, not just a scoping note.
  - *A corrupted leader has no self-repair path*: reconciliation is follower-to-leader only; there is no authority to check the leader against without majority-vote verification, which was out of scope.
  - *Unbounded history*: every past version's full content is kept forever (needed as a merge base) and tombstones are never garbage-collected.

9 Future works

- **Highly-available gateway**

  Run two or more stateless gateway instances behind a virtual IP or a small load balancer, or give the client the replica list as a discovery fallback so a gateway outage is survivable.

- **Follower reads**

  Allow reads from followers once they can prove freshness: either lease-based reads from the leader, or a read-index barrier, or serving a follower read only when its applied index is within a bound of the leader's commit index. This would let read throughput scale with the cluster.

- **Leader integrity checking**

  Add a periodic cross-replica hash comparison with majority vote, so a corrupted leader can be detected and forced to step down and re-sync, closing the one gap in the current self-healing story.

- **Security**

  TLS termination at the gateway, an API token or mutual TLS for clients, mutual TLS between replicas, HMAC-signed reconciliation payloads, and per-note ownership/ACLs.

- **History management**

  Garbage-collect old `file_versions` beyond a retention window (keeping only versions that could still be a merge base), and garbage-collect tombstones after all replicas have durably seen them.

- **Larger notes in the TUI**

  A scrolling editor so the window-fit restriction can be lifted.

References

Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.