Dok

A Distributed, Real-Time Collaborative Document Editor

Final Report for the Distributed Systems Course 2025/2026

Lorenzo Tosi
lorenzo.tosi10@studio.unibo.it

Alessandro Stefani
author2@gmail.com

June 19, 2026

Dok is a distributed, real-time collaborative document editing platform implemented as a web application. It allows multiple users to create, share, and simultaneously edit text documents. The system is built on a client–server architecture in which the backend is horizontally scaled across two **Node.js/Express** server instances, load-balanced by an **Nginx** reverse proxy. Real-time collaboration is achieved through **Yjs** Conflict-free Replicated Data Types (CRDTs) transported over **Socket.io** WebSocket connections; cross-instance event propagation is coordinated via a shared **Redis** Pub/Sub adapter, which also enables cluster-wide user-presence tracking. Persistent state is stored in a **MongoDB** document database. Documents are organised in folders and can be made public or private, with sharing policies (*viewer*, *commenter*, *editor*) enforced through **JWT**-based authentication and role-based access control. An administrative dashboard provides real-time platform-wide statistics, audit trails, user presence information, and site-access logs. The entire infrastructure is containerised with **Docker Compose**, making deployment reproducible with a single command.

Disclaimer

During the preparation of this work, the author(s) used an AI-assisted coding assistant to support code scaffolding, refactoring, and the drafting of parts of this report.

After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

1.1 Type of Product

Dok is a **web application with a graphical user interface**, accessible from any modern browser without additional software installation. It combines a single-page application (SPA) frontend, built with Vue 3 and Vite, with a horizontally scalable REST and WebSocket backend.

1.2 Use-Case Description

Dok targets teams of knowledge workers—students, researchers, and professionals—who need to collaborate on written documents in real time from geographically dispersed locations.

What the system does. Users create rich-text documents (formatted paragraphs, headings, inline styling) and folders to organise them. They can keep documents private, publish them to a global feed, or share them selectively with other registered users under a specific permission level. Multiple users who have access to the same document can edit it concurrently: their changes converge automatically through CRDT semantics, and their cursor positions are broadcast live to collaborators. Inline comments allow asynchronous annotation without disrupting the editing flow. An integrated AI assistant (backed by Google Gemini) can rewrite any selected passage following a free-text instruction. An administrator role provides platform-wide oversight: real-time user-presence, per-user file-system trees, usage statistics, audit logs of document accesses and modifications, and site-access logs.

Where and how users interact. Users interact exclusively through a browser-based SPA. They may be physically anywhere; the platform is designed with network-distributed access in mind. Authenticated users access private and shared content; unauthenticated visitors can read public documents and browse the public dashboard.

User roles. Three end-user roles exist: *Administrator* (first registered user is automatically promoted), *Authenticated User*, and *Guest* (unauthenticated). Document-level collaboration roles further distinguish *owner*, *editor*, *commenter*, and *viewer*.

Data stored. The system stores user accounts, documents (rich-text content serialised both as Yjs binary state and Tiptap JSON), folders, comments, inter-user sharing metadata, per-document audit logs, site-access logs, and real-time notifications.

1.3 Why Distribution Is Needed

Distribution addresses several independent needs:

- **Concurrency and real-time collaboration.** Multiple users may edit the same document simultaneously. A single-process server would create a bottleneck and a single point of failure; horizontal scaling across multiple server instances, coordinated by Redis, removes both obstacles.
- **Fault tolerance.** Running at least two backend replicas behind a load balancer ensures that the failure of one instance does not render the service unavailable.
- **Resource sharing.** All backend instances share a single MongoDB database and a single Redis broker, making the system's data consistent regardless of which replica serves a given request.
- **Scalability.** The Nginx hash-based routing combined with the Redis Pub/Sub adapter allows new backend instances to be added with minimal configuration changes.

2 Requirements Elicitation and Analysis

2.1 Glossary

Dok The name of the platform and the informal term for a document managed within it.

CRDT Conflict-free Replicated Data Type—a data structure that can be merged from multiple concurrent sources without conflicts.

Yjs state The binary serialisation of a Yjs document, stored in MongoDB and used to restore in-memory state when a document room is created.

Room A Socket.io room identified by a document ID; all collaborators of a document join this room.

Presence Whether a registered user currently has at least one active WebSocket connection to any server instance.

Audit log A record of a user's access or modification event on a document.

2.2 Functional Requirements

FR-1 User Registration and Authentication. The system shall allow a visitor to register with email, first name, last name, and password. Subsequent logins shall return a JWT. The first registered user shall automatically receive the *Administrator* role. *Acceptance criterion:* A POST to `/api/auth/register` returns HTTP 201 with a valid JWT; a duplicate email returns HTTP 409.

FR-2 Document Lifecycle. An authenticated user shall be able to create, rename, and delete documents. Public documents shall be visible to all (including guests);

private documents shall be visible only to their owner and explicitly invited collaborators. *Acceptance criterion:* CRUD operations on `/api/documents` succeed for authorised users and return HTTP 403/404 for unauthorised ones.

FR-3 Folder Hierarchy. Users shall be able to organise documents in nested folders with configurable visibility. *Acceptance criterion:* Folders support a self-referential parent/child relationship; documents within a folder are filtered when querying with `parentId`.

FR-4 Fine-Grained Document Sharing. The document owner shall be able to share a document with any registered user by email, assigning a role of *viewer*, *commenter*, or *editor*. Roles can later be updated or revoked. *Acceptance criterion:* PUT `/api/documents/share` modifies the `sharedWith` array; the target user receives a real-time notification.

FR-5 Real-Time Collaborative Editing. Concurrent edits by multiple users to the same document shall converge without loss of data, using CRDT semantics. All collaborators shall see changes within sub-second latency. *Acceptance criterion:* Two clients connected to *different* server instances who join the same document room both receive the `crdt-update` events produced by the other (validated in `distributed.test.ts`).

FR-6 Cursor Awareness. Users editing a document shall see the live cursor positions and names of co-editors. *Acceptance criterion:* Clients receive `awareness-update` events relayed by the server.

FR-7 Comments. Owners, editors, and commenters shall be able to add inline comments to a document. Owners and editors can delete any comment; the comment author can delete their own. *Acceptance criterion:* Verified by the role-matrix tests in `comments.test.ts`.

FR-8 AI Writing Assistance. An authenticated editor/owner shall be able to select a passage and submit a free-text instruction. The system shall invoke Google Gemini 2.5 Flash and return the rewritten text. *Acceptance criterion:* POST `/api/llm` returns a JSON with `originalText` and `rewrittenText`.

FR-9 Real-Time Notifications. Users shall receive in-app notifications (via WebSocket) when a document is shared with them, when their permissions change, or when a document they have access to is deleted. *Acceptance criterion:* Notification documents are persisted in MongoDB; unread count is updated in real time.

FR-10 Administrative Dashboard. An administrator shall be able to view: all registered users with online/offline presence; per-user usage metrics (documents created, shared); a user's private/public file-system tree; per-document audit logs; global platform statistics; site-access logs. The administrator shall also be able to promote/demote user roles. *Acceptance criterion:* Admin endpoints return HTTP 403 for non-admin tokens.

FR-11 Automatic Document Persistence. The system shall automatically save the Yjs document state to MongoDB when editing activity ceases (debounced 3-second timer) and unconditionally when the last collaborator leaves. *Acceptance criterion:* Verified in `distributed.test.ts`.

FR-12 API Documentation. The backend shall expose interactive API documentation via Swagger UI at `/api-docs`. *Acceptance criterion:* Verified in `swagger.test.ts`.

2.3 Non-Functional Requirements

NFR-1 Availability. The service shall remain available to clients if one of the two backend instances crashes, relying on Nginx's upstream failure detection.

NFR-2 Consistency of Collaborative State. Concurrent edits to the same document shall always converge to an identical final state, irrespective of the order in which updates are applied, thanks to CRDT properties.

NFR-3 Low Editing Latency. Changes made by one collaborator shall be propagated to other clients with sub-second latency under normal network conditions.

NFR-4 Scalability. The architecture shall support the addition of further backend replicas without changes to application code.

NFR-5 Security. Passwords shall be stored as bcrypt hashes (cost factor 12). All protected endpoints shall require a valid JWT; admin endpoints shall additionally verify the *ADMIN* role.

NFR-6 Maintainability. The codebase shall be modular (controllers, services, models, socket handlers) and fully typed with TypeScript.

2.4 Implementation Constraints

IC-1 The backend shall be implemented in **TypeScript** on **Node.js**, chosen for the ecosystem's excellent WebSocket support and the team's proficiency.

IC-2 The frontend shall be a **Vue 3** SPA, chosen for its reactive component model and first-class TypeScript integration.

IC-3 Real-time collaboration shall be built on **Yjs** CRDTs, the de-facto standard for operation-based collaborative editing without a central OT server.

IC-4 The entire backend stack shall be containerised with **Docker** to enable one-command deployment and reproducible environments.

IC-5 The AI feature shall use the **Google Gemini** API (`gemini-2.5-flash` model), requiring a valid `GEMINI_API_KEY` environment variable.

2.5 Relevant Distributed System Features

Transparency. Access, location, and replication transparency are important. Clients connect to a single Nginx endpoint and are unaware of which backend instance serves their requests. The Redis adapter makes socket rooms appear global; a client on server 1 receives events emitted by server 2 without any application-level code change.

Fault tolerance and availability. Two backend replicas guarantee that a single-instance crash does not interrupt the service. A 3-second grace period in the `PresenceManager` avoids misclassifying brief network drops as permanent disconnections (network flap handling). The debounced save mechanism and the unconditional flush on last-client-leave provide durability even if a server crashes mid-session.

Scalability. The hash-consistent routing policy in Nginx routes all WebSocket connections for the same document to the same backend instance (based on the `documentId` query parameter), minimising unnecessary Redis roundtrips. Additional instances can be added to the `upstream` block and the Docker Compose file.

Security and trust. Passwords are bcrypt-hashed, all API access is JWT-gated, and document-level RBAC enforces four distinct permission levels. Admin-only endpoints are double-guarded by both the `requireAuth` and `requireAdmin` middlewares.

Resource sharing. MongoDB and Redis are shared singletons across all backend replicas, providing a single authoritative state for documents, users, and socket room membership.

Openness and interoperability. The REST API is documented with OpenAPI/Swagger, enabling third-party integration. The Tiptap JSON format stored alongside the Yjs binary state allows export to standard rich-text formats in future integrations.

Performance and concurrency. Yjs CRDTs handle concurrent writes without locking. MongoDB's optimistic concurrency (the `optimisticConcurrency` flag on Folder and Document schemas) handles rare database-level write conflicts. Audit-log aggregation (character counting) is batched in memory and flushed lazily to avoid per-keystroke database writes.

Economy. The use of free-tier-compatible managed services (MongoDB, Redis) in the Docker Compose setup keeps operational costs minimal for small deployments.

3 Design

This chapter presents the strategies used to meet the requirements of section 2.

3.1 Architecture

Dok follows a classic **client–server, three-tier architecture** with a *stateless application tier* and a *shared state tier*:

- **Presentation tier** — a Vue 3 SPA running in the user's browser.
- **Application tier** — one or more identical Node.js processes exposing both a REST API (Express) and a WebSocket endpoint (Socket.IO). This tier is *stateless with respect to HTTP* but holds *ephemeral session state* for the documents currently being edited (the in-memory Yjs documents).
- **State tier** — a single MongoDB instance for persistence and a single Redis instance used as a Socket.IO pub/sub backbone.

The chosen style is a *service-oriented, horizontally scalable monolith*: each backend instance is a replica of the same application, and the load balancer distributes requests among them. This style is appropriate because the dominant workload (live editing) is interactive and benefits from low-latency, sticky connections, while the REST workload is stateless and trivially parallelisable. Distribution is achieved by *replication of the application code* rather than by partitioning the domain across microservices, which keeps the system simple while still allowing scale-out.

A central design decision is **where edit consistency is enforced**. Rather than using a central sequencer or operational-transform server, Dok delegates consistency to a CRDT: each client and the server hold a replica of the document's Yjs document, and updates commute. This removes the need for distributed locking and is the key enabler of location transparency.

3.2 Infrastructure

The infrastructural components, as defined in the `docker-compose.yml`, are:

- **Nginx** (`nginx:alpine`): A single entry point on port 3000. It terminates incoming HTTP and WebSocket connections and forwards them to the upstream cluster. WebSocket upgrade headers (`Upgrade`, `Connection`) are forwarded explicitly. The real client IP is passed via `X-Real-IP`.
- **backend1** and **backend2**: two replicas of the Node.js application, built from `server/Dockerfile`. They are intentionally identical to demonstrate horizontal scaling.
- **mongo** (`mongo:latest`): the single shared document database, exposed on port 27017.
- **redis** (`redis:alpine`): the pub/sub broker for Socket.IO cross-instance messaging, exposed on port 6379.

Distribution over the network. In the reference deployment all containers run on a single Docker host (single datacenter, single network). However, the topology is designed so that any subset of these components could be relocated: the backends are stateless and discover the database and broker purely through environment variables (`MONGO_URI`, `REDIS_URL`), so placing MongoDB or Redis on another host or a managed cloud service requires no code change.

Component naming and discovery. Discovery is centralised and static: Docker Compose's internal DNS resolves service names (`backend1`, `backend2`, `mongo`, `redis`) to their container IP addresses. Nginx references the two backends by name in its `upstream` block; the Node.js processes reference MongoDB and Redis by URL. There is no dynamic service registry because the set of backends is fixed and small.

Routing strategy. This is one of the most important infrastructural choices. The `nginx.conf` defines a single upstream and uses the `hash` directive with the `consistent` option:

```
map $arg_documentId $route_key {
    "~^.+$"  $arg_documentId;
    default  $remote_addr;
}
upstream websocket_cluster {
    hash $route_key consistent;
    server backend1:3000;
    server backend2:3000;
}
```

The hash key is the `documentId` query parameter when present, and the client IP otherwise. This implements **application-aware consistent hashing**: all clients editing the *same* document are routed to the *same* backend, which is where that document's live Yjs replica resides. Consistent hashing (rather than round-robin) is chosen so that adding or removing a backend only re-maps a fraction of the keyspace, minimising disruption. Pure stateless REST calls (no `documentId`) fall back to client-IP hashing for a rough load balance. Nginx is configured for WebSocket upgrade (`proxy_http_version 1.1` plus the `Upgrade`/`Connection` headers), which is required for Socket.IO long-lived connections.

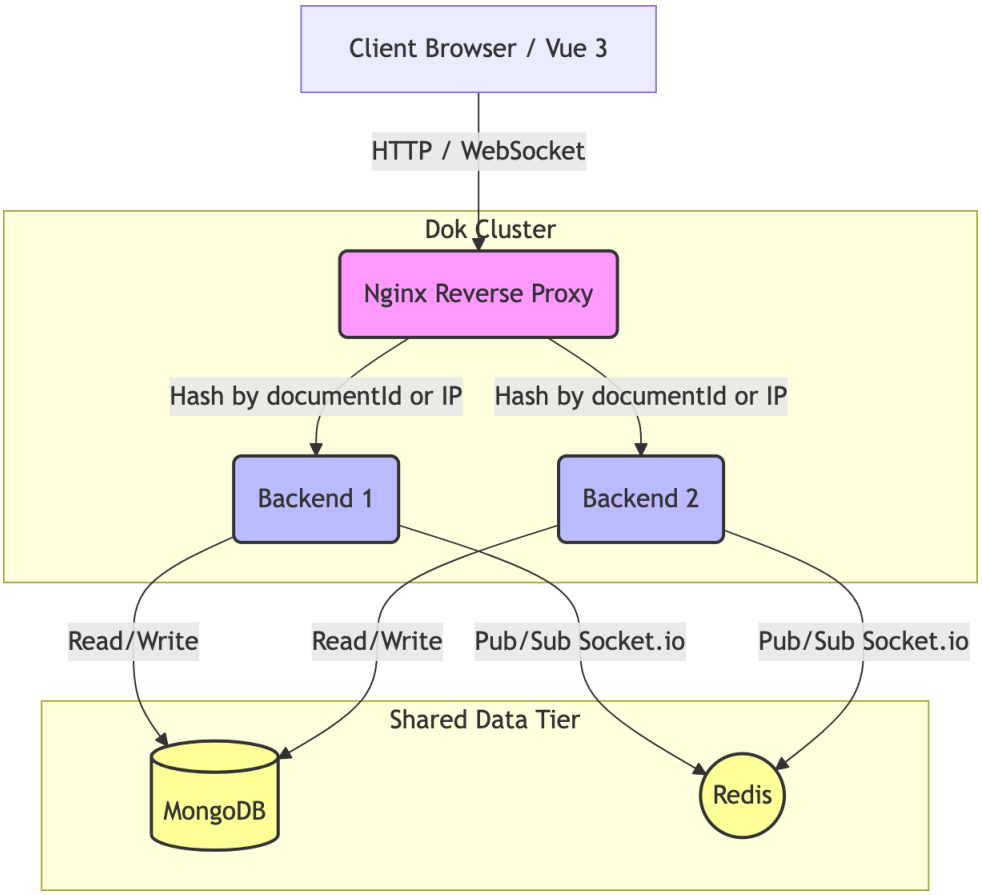

Figure 1: Component and Deployment Diagram.

3.3 Modelling

Domain entities. The persisted domain model, expressed as Mongoose schemas, consists of:

- `User` — `email` (unique), `passwordHash`, `role` (`USER`—`ADMIN`), `firstName`, `lastName`, `lastSeen`, plus timestamps.
- `Document` — `title`, `ownerId`, `folderId`, `visibility` (`private`—`public`), an array `sharedWith` of {`userId`, `role`}, an array `comments` of {`userId`, `content`}, the binary `yjsState` (the serialised CRDT), and a `tiptapJson` projection for rendering.
- `Folder` — `name`, `ownerId`, `parentId`, `visibility`.
- `Notification` — `recipient`, `sender`, `type` (`SHARE`—`PERM_CHANGE`—`DOCUMENT_DELETE`—`SYSTEM`), `title`, `message`, `documentId`, `read`, `link`.

- `AuditLog` — `documentId`, `userId`, `type` (`access`—`modification`), `charactersInserted`.
- `SiteAccessLog` — `userId`, `loginAt`, `logoutAt`, `ipAddress`.

In-memory state. Alongside the persisted model, each backend holds a `Map<string, ActiveDocState>` of documents currently being edited. Each entry contains the live `Y.Doc`, a `clientsCount`, a debounced `saveTimeout`, and a `pendingUserChars` map used to aggregate modification audit logs. This ephemeral state is the unit that is pinned to a single backend by the consistent-hashing router.

Mapping entities to infrastructure. Persistent entities live in the shared MongoDB, accessible from every backend. The live editing state (the CRDT) lives in the memory of exactly one backend per document — the one selected by Nginx. Awareness (caret positions, colour, user name) is ephemeral and not persisted; it flows only through the socket layer.

Domain events. Examples include: *user registered/logged in*, *document created/deleted/renamed*, *document shared/unshared*, *comment added/deleted*, *user joined/left a document*, *role changed*.

Messages. The system exchanges two families of messages:

- **REST commands/queries** (HTTP/JSON): `POST /api/auth/register`, `POST /api/documents`, `PUT /api/documents/share`, `POST /api/documents/:id/comments`, the admin endpoints, etc. These are the source of truth for non-collaborative state changes.
- **Socket.IO events**: `join-document`, `sync-document`, `crdt-update`, `awareness-update`, `leave-document`, plus notification events (`new-notification`, `document-renamed`, `presence_update`, `new_audit_log`, ...). These carry the live, collaborative traffic.

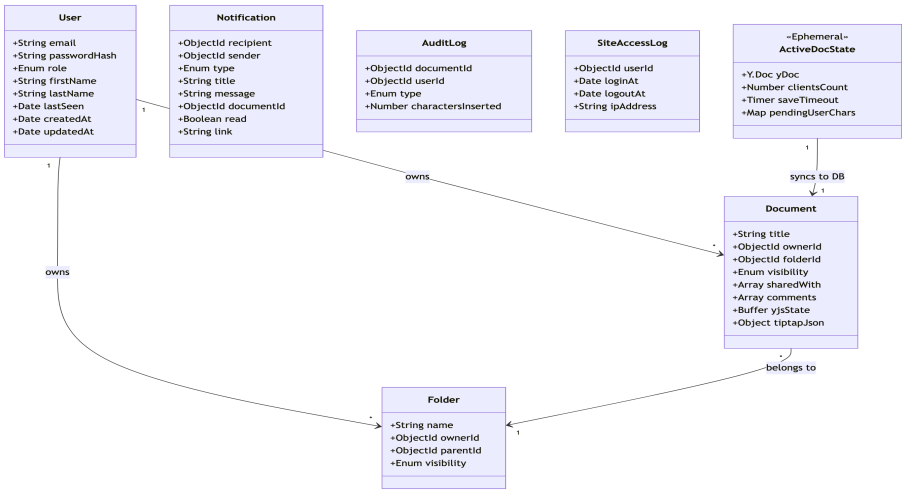

Figure 2: Class diagram.

3.4 Interaction

Two distinct interaction patterns coexist, chosen to match the nature of each operation:

- **Request–response (synchronous, over HTTP/JSON).** Used for operations that change durable state and require an immediate success/failure acknowledgement: authentication, CRUD on documents and folders, sharing, commenting, and all administrative actions. This pattern is stateless and therefore trivially load-balanced across backends by Nginx.
- **Publish–subscribe / event streaming (asynchronous, over WebSocket).** Used for the live editing session and for real-time notifications. Within a single backend, the classic Socket.IO room abstraction is used: joining a document means joining a room named after the document id, and `socket.to(room).emit(...)` fans updates out to the other participants. *Across* backends, the Socket.IO Redis adapter transparently relays emissions through Redis pub/sub channels, so a client connected to backend2 receives an update emitted on backend1 as if they were on the same process.

The lifecycle of a collaborative editing session is the canonical interaction:

1. The client opens a document, opening a Socket.IO connection whose URL carries `?documentId=...`; Nginx hashes this id and routes the connection to a fixed backend.
2. The client emits `join-document`; the server loads (or reuses) the in-memory CRDT, applies any persisted binary state, joins the socket room, and replies with `sync-document` containing the full encoded CRDT state.
3. On each local edit the client emits `crdt-update` with the binary update; the server applies it to its CRDT (tagging the transaction origin with the user id for auditing) and re-broadcasts it to the room via `socket.to(documentId).emit(...)`; the Redis adapter forwards it to other backends if needed.
4. Awareness updates (caret position, selection, user identity) flow symmetrically through `awareness-update`.
5. On `leave-document` or `disconnecting`, the server decrements the client count; when it reaches zero the final state is flushed to MongoDB and the in-memory CRDT is garbage-collected.

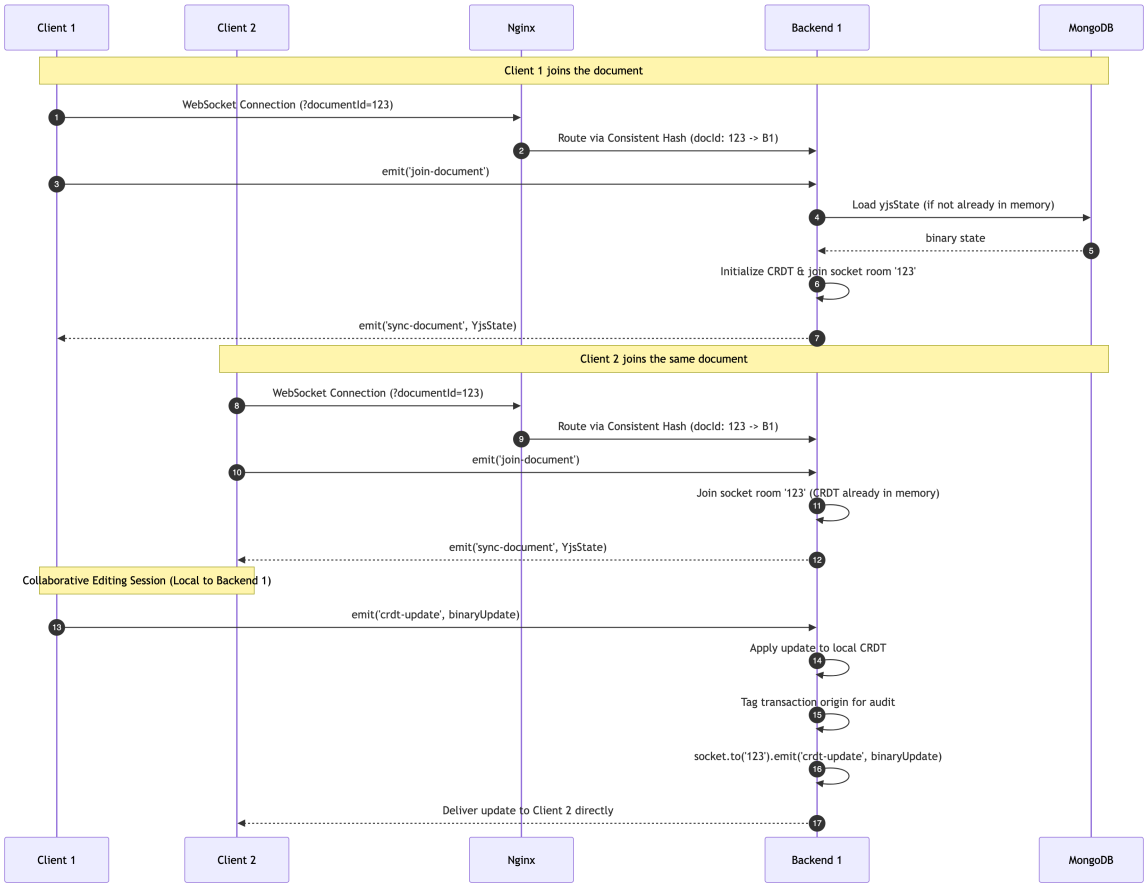

Figure 3: Example of 2 user joining and editing the same document.

Real-time file-system event propagation. Beyond the collaborative editing channel, Redis plays a second, equally important role: it is the backbone for propagating *structural changes* to the file system in real time. When a user performs an action that modifies the shared state of the dashboard—creating, renaming, or deleting a document, or sharing/unsharing it with a collaborator—the service layer invokes the `NotificationManager`, which emits targeted Socket.IO events to one or more rooms. Because the Socket.IO Redis adapter is active, these emissions are transparently forwarded to *all* backend instances, so every connected client receives the update regardless of which server instance hosts their WebSocket connection.

The rooms involved and the events emitted are:

- `global-dashboard` — receives `global-document-created`, `global-document-renamed`, and `global-document-deleted` whenever a *public* document changes. All clients watching the public feed join this room (via `join-public-dashboard`) and refresh their view without polling.

- `user:{id}` (private per-user room) — receives `private-document-created`, `private-document-deleted`, `document-renamed`, `document-shared`, `document-unshared`, `new-notification`, and `role-changed`. Every authenticated client joins their own private room immediately on connection (`socket.join(‘user:${userId}‘)`), so the owner of a document sees their dashboard updated instantly when one of their private documents is renamed or deleted, and a collaborator's "Shared with me" panel updates immediately when a document is shared with them or revoked.

- {`documentId`} (document editing room) — receives `document-renamed`, `document-shared`, `document-unshared` while users are actively inside the editor, so the title bar and the collaborator list react in real time without a page reload.

- `admin:dashboard` — receives `presence_update`, `user_metrics_update`, and `new_audit_log`. The administrator panel joins this room and displays live counters (documents created, shared) and the online/offline state of every user.

The entire notification chain is synchronous from the service layer's perspective: after each mutating REST call completes (e.g., `PUT /api/documents/share`), the service calls `NotificationManager.notifyDocumentShared(...)`, which calls `io.to(...).emit(...)` on the local Socket.IO server instance. The Redis adapter serialises the emission and publishes it to the appropriate Redis channel; every other backend instance subscribed to that channel delivers it to its locally connected clients. The result is that the client who *initiated* the REST action and the clients who are affected by it—even if they are connected to *different* backend instances—all observe the updated state within milliseconds, without any polling.

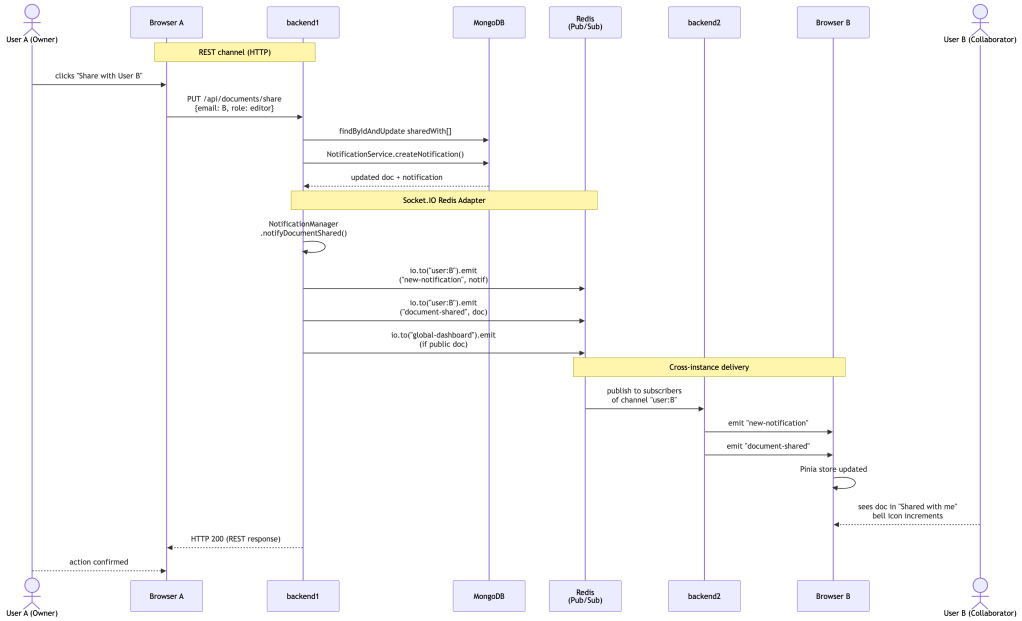

Figure 4: Example of real time creation of new document, sharing it and receiving the notification.

3.5 Behaviour

Backend behaviour. Each backend is event-driven and single-threaded (Node.js). Its components behave as follows:

- *Document handler* (stateful, per document). On `join-document` it lazily loads the CRDT from MongoDB and registers a deep observer on the Yjs XML fragment. The observer inspects each CRDT transaction's `origin` (which the server sets to the editing user's id when applying a remote update) and accumulates inserted characters per user into `pendingUserChars`; this provides attribution for the audit log without per-keystroke database writes. On every `crdt-update` it resets a 3-second debounce timer: when editing pauses for 3 seconds, the full binary state and the Tiptap JSON projection are persisted. This *debounced persistence* is the main mechanism that keeps database write load proportional to *editing sessions* rather than to *keystrokes*, addressing the performance concern of section 2.5.

- *Presence manager* (stateful, per user). On connection it joins a per-user room `user:<id>`; because the room is shared across backends through Redis, the count of sockets in the room is a cluster-wide online indicator. On disconnect it waits a 3-second *grace period* before declaring the user offline: if a new connection appears within the window (a transient network flap or a device switch) the offline transition is cancelled. Only when the room is genuinely empty does it update `lastSeen` in the database and emit `presence_update` to the admin room.

- *Notification manager* (stateless facade over the socket instance). It is initialised once at startup with a reference to the Socket.IO `Server` object and exposes a set of domain-level methods called by the service layer after each mutating operation:
  - `notifyDocumentCreated`.
  - `notifyDocumentDeleted`.
  - `notifyDocumentRenamed`.
  - `notifyDocumentShared`.
  - `notifyDocumentUnshared`.
  - `notifyCommentAdded`.
  - `notifyCommentDeleted`.
  - `notifyAdminMetricsUpdate`.
  - `notifyRoleChanged`.

  Each method translates a domain event into one or more `io.to(room).emit(...)` calls, routing the event to the correct set of recipients (a single user room, a document room, the global dashboard, or the admin dashboard). Because the Socket.IO Redis adapter is attached to the same `io` instance, every emission is automatically replicated across all backend instances: the `NotificationManager` is therefore completely stateless and instance-agnostic—it emits locally and Redis takes care of cluster-wide delivery.

- *Auth middleware* (stateless). Validates the JWT on every protected HTTP route and on the socket handshake; for sockets, an absent token is tolerated (guest), while an invalid token is rejected.

Who updates the state. The single source of truth for a document's content is the in-memory CRDT on the owning backend; it is updated by the `crdt-update` handler and periodically checkpointed to MongoDB by the debounce/flush logic. All other durable state (users, folders, sharing, notifications, logs) is updated synchronously by the service layer in response to REST commands, against the shared MongoDB. Because the CRDT guarantees convergence, no component needs to coordinate with others to "update the state correctly" — replicas simply apply commutative updates.

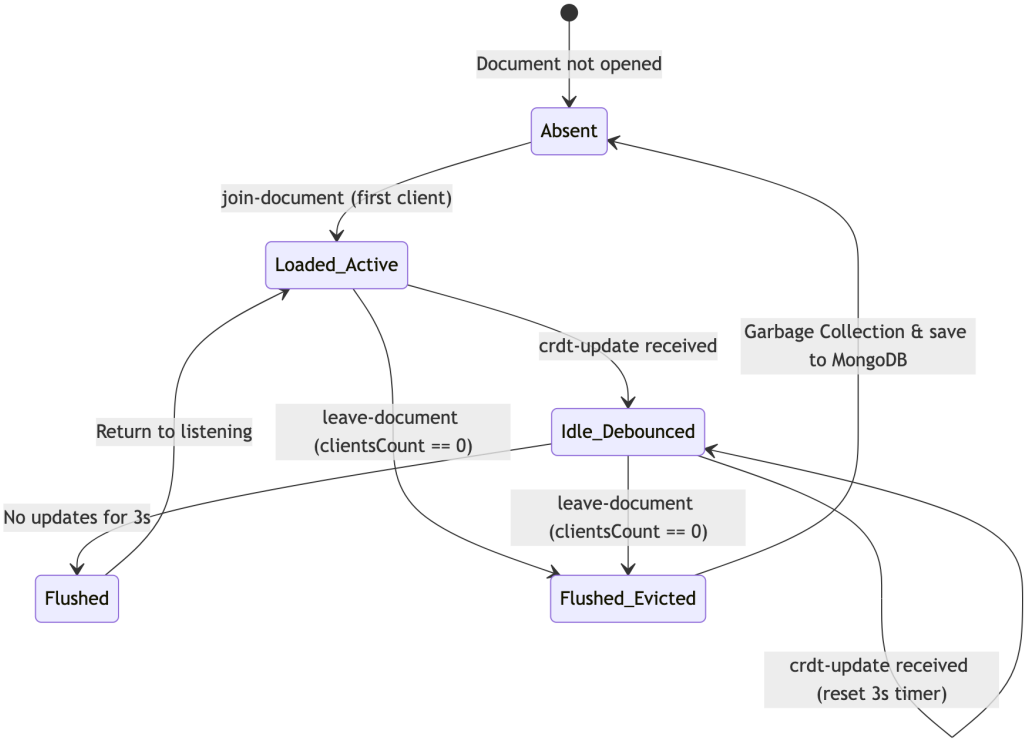

Figure 5: Document lifecycle inside server.

3.6 Data and Consistency Issues

Persistent data. All domain entities are stored in MongoDB as BSON documents. The choice of a document store is motivated by the document-centric nature of the application: each Dok naturally maps to a MongoDB document with embedded comments and sharing metadata, avoiding expensive JOIN-equivalent operations.

CRDT consistency. Document content is maintained as a Yjs `Y.Doc`. Yjs implements a LOGOOT-inspired CRDT with vector-clock based causality. Because CRDT merges are commutative, associative, and idempotent, concurrent edits from any number of clients always converge to the same final state without requiring a central coordinator or Operational Transformation. The server holds the canonical in-memory CRDT state for each active document and serialises it to MongoDB on a periodic basis.

Optimistic concurrency. The `Document` and `Folder` Mongoose schemas use `optimisticConcurrency: true`, which adds a `__v` (version key) check on `save()` to detect conflicting writes at the database level (analogous to optimistic locking in relational databases). This guards against rare race conditions in HTTP endpoints that read-then-modify a document.

Shared data between components. The `yjsState` field in MongoDB acts as the hand-off point between instances. If a client connects to instance 2 for a document that instance 1 last persisted, instance 2 loads the latest `yjsState` from MongoDB and reconstructs the `Y.Doc` from scratch, correctly replaying all prior operations encoded in the binary snapshot.

3.7 Fault-Tolerance

Replica redundancy. Two backend instances eliminate the single point of failure at the application server level. If one instance crashes, Nginx's upstream health-check mechanism stops routing new connections to it, and clients will reconnect to the surviving instance. The Redis adapter state (room membership) reconstructs automatically as clients reconnect.

Grace period for network flap. The `PresenceManager.removeConnection` method waits 3 seconds before marking a user offline and updating `lastSeen`. This handles transient network interruptions and multi-tab scenarios: if the socket reconnects within the grace period, the user is never declared offline.

Debounced persistence and flush-on-close. The dual save strategy (3-second debounce *plus* unconditional flush when `clientsCount` reaches zero) ensures that the maximum data loss window is bounded by the debounce interval. Even a sudden server crash at most loses 3 seconds of edits for documents that had active clients on that instance at the time of failure.

LLM retry logic. The `LLMEngine` retries failed Gemini API calls up to 3 times before surfacing an error, tolerating transient network or rate-limit failures from the external API.

Error handling. Errors in background tasks (metric updates, sharing metric refreshes) are caught and logged but do not propagate to the calling request, preventing auxiliary failures from disrupting the primary user action.

3.8 Availability

Caching. There is intentionally little caching: the in-memory `activeDocuments` map is effectively a *session cache* of live documents, present only on the owning backend. The `tiptapJson` projection persisted alongside the CRDT acts as a pre-computed rendering cache so that read-only opens do not need to run the Yjs→Tiptap transformation.

Load balancing. Nginx performs load balancing with the consistent-hashing strategy described in section 3.2. This simultaneously balances connection load and enforces session stickiness, which is what makes the single-replica-per-document design correct. For pure REST traffic (no `documentId`), client-IP hashing provides a fairer distribution.

Network partitioning. Because all backend instances share MongoDB and Redis, a partition between the two instances would be detected by the Redis adapter (connection errors logged), but MongoDB reads/writes from each instance would still succeed independently. The system makes no formal partition-tolerance guarantees; it prioritises consistency and availability under the assumption that intra-datacenter network partitions are rare.

3.9 Security

Authentication. All protected REST endpoints require an `Authorization: Bearer {JWT}` header. The Socket.io handshake passes the JWT via `socket.handshake.auth.token`. Tokens are signed with the HS256 algorithm using a secret read from the `JWT_SECRET` environment variable (defaulting to a development placeholder that must be overridden in production). Tokens expire after 24 hours.

Password security. The `User` model hashes passwords via a Mongoose pre-save hook using bcrypt with a cost factor of 12. The `passwordHash` field has `select: false`, preventing accidental exposure in query results.

Authorisation. Three layers of access control are enforced:

1. **Endpoint-level RBAC**: `requireAuth` (any valid JWT) and `requireAdmin` (JWT with `role === ADMIN`) middleware guard route groups.
2. **Document-level ownership checks**: Service methods verify `ownerId` or `sharedWith` membership before performing mutations.
3. **Comment-level role checks**: Comment creation and deletion enforce the four document roles (owner, editor, commenter, viewer).

Input validation. The `requireBodyField` and `validateMongoIdParam` middleware functions reject malformed requests before they reach business logic.

Admin data masking. The `AdminService.getDocumentInfoForAdmin` method replaces the title of private documents with the string "Documento Riservato", and strips `tiptapJson` and `yjsState` from the response, ensuring that administrators cannot read the *content* of private documents they do not own.

4 Implementation

4.1 Network Protocols

- **HTTP/1.1** for REST API traffic. Nginx upgrades WebSocket connections via the `Upgrade` header mechanism.

- **WebSocket** (over HTTP/1.1 Upgrade) for all real-time communication, managed by Socket.io.
- **Redis RESP** (Redis Serialisation Protocol) for pub/sub communication between the Socket.io adapter clients and the Redis broker.
- **MongoDB Wire Protocol** for all database interactions.

4.2 Data Representation

- **REST API**: All request and response bodies are JSON; Express's `express.json()` middleware handles parsing and serialisation.
- **Yjs CRDT state**: Transmitted over WebSocket as binary `Uint8Array` (encoded by `Y.encodeStateAsUpdate`); stored in MongoDB as a `Buffer` in the `yjsState` field.
- **Rich-text snapshot**: A Tiptap-compatible JSON document (`tiptapJson`) is stored alongside the binary state for potential future use by read-only consumers.
- **JWT payload**: JSON object with `id` and `role` fields, signed with HS256.

4.3 Database

MongoDB is queried via **Mongoose** ODM (version 9). All domain entities are defined as strongly-typed Mongoose schemas with TypeScript interfaces extending `Document`. Indexes are declared at the schema level to support frequent query patterns. The database name is `dok` (configurable via `MONGO_URI`).

4.4 Authentication and Authorisation

- **Authentication**: Bearer JWT (library: `jsonwebtoken` v9, algorithm: HS256, expiry: 24h).
- **Authorisation**: Role-Based Access Control (RBAC). Global roles (`USER`, `ADMIN`) are checked at the middleware layer. Document roles (`owner`, `editor`, `commenter`, `viewer`) are checked at the service layer.

4.5 Technological Details

Backend.

- **Runtime**: Node.js 22 (LTS); **language**: TypeScript 6.
- **HTTP framework**: Express 5.
- **WebSocket**: Socket.io 4.8 with `@socket.io/redis-adapter` 8.3.

- **CRDT**: Yjs 13.6 (in-process `Y.Doc`) + `@hocuspocus/transformer` for Yjs↔Tiptap JSON conversion.
- **Database client**: Mongoose 9.
- **Cache/broker**: `redis` 5 client library.
- **Password hashing**: bcrypt 6 (cost factor 12).
- **JWT**: `jsonwebtoken` 9.
- **AI integration**: `@google/generative-ai` 0.24 (Gemini 2.5 Flash).
- **API documentation**: `swagger-jsdoc` 6 + `swagger-ui-express` 5.
- **Development runner**: `tsx` 4.

Frontend.

- **Framework**: Vue 3.5 (Composition API) with TypeScript.
- **Build tool**: Vite 8.
- **State management**: Pinia 3.
- **Routing**: Vue Router 5.
- **Rich-text editor**: Tiptap 3 with extensions: `collaboration` (Yjs integration), `collaboration-caret` (cursor awareness), `bubble-menu`, `highlight`, `text-align`, `font-family`, `color`, `underline`, `history`.
- **CRDT client**: Yjs 13.6 + `y-protocols`.
- **HTTP client**: Axios 1.15.
- **WebSocket client**: `socket.io-client` 4.8.
- **Charts**: ApexCharts 5 via `vue3-apexcharts`.

Infrastructure.

- **Containerisation**: Docker; Docker Compose v2 for service orchestration.
- **Reverse proxy / load balancer**: Nginx Alpine.
- **Database**: MongoDB (`mongo:latest` image).
- **Broker**: Redis (`redis:alpine` image).

5 Validation

5.1 Automatic Testing

Tests are written with **Vitest** 4 and executed with `npm test` from the `server/` directory. The test suite uses `mongodb-memory-server` 11 and `redis-memory-server` 0.16 to run in-process, fully isolated database instances, requiring no external infrastructure.

Test files and rationale.

`tests/setup.ts` Shared helpers: `connectDBForTesting`, `disconnectDBForTesting`, `clearDBForTesting`. Each test file clears all collections in `beforeEach` to guarantee test isolation.

`tests/auth.test.ts` **Unit/integration tests for FR-1 and NFR-5.**

- Tests successful registration (HTTP 201, JWT returned, password not exposed, user saved in DB).
- Tests duplicate-email rejection (HTTP 409).
- Tests login with correct and wrong credentials.
- Tests the `requireAuth` middleware: accepts valid tokens, rejects missing tokens (HTTP 401), rejects malformed tokens (HTTP 401).

Uses `supertest` to fire real HTTP requests against the Express app instance.

`tests/comments.test.ts` **Integration tests for FR-7.** A matrix of role × action tests:

- Owner, editor, commenter can add comments (HTTP 201).
- Viewer and unauthenticated stranger cannot add comments (HTTP 403).
- Comment creator, document owner, and document editor can delete a comment.
- A commenter cannot delete another commenter's comment (HTTP 403).
- Viewer cannot delete any comment (HTTP 403).

`tests/distributed.test.ts` **Integration tests for the distributed-systems core (FR-5, FR-11, NFR-1, NFR-2).** This is the most complex test file. It instantiates *two real Socket.io servers* (on ports 3001 and 3002) sharing a real Redis in-memory instance, simulating the production multi-instance topology.

- **Redis Adapter verification**: asserts that both `io1` and `io2` use the `RedisAdapter` class.
- **Presence propagation**: a client connects to server 1; `PresenceManager.isUserOnline` (querying through the Redis adapter of server 2) returns `true`. After disconnection and the 3.5-second grace period, the user is correctly marked offline and `lastSeen` is updated in MongoDB.

- **CRDT synchronisation across instances**: client A connects to server 1, client B to server 2; both join the same document room. Client A emits a `crdt-update`; the test asserts that client B receives the identical binary update, proving cross-instance propagation via Redis.
- **Persistence on disconnect**: a client edits a document and disconnects; the test waits and then fetches the document from MongoDB, verifying that `yjsState` is non-empty and that the content reconstructed from it matches the original text ("Hello Auto-Save!").

`tests/swagger.test.ts` Confirms that the Swagger UI endpoint returns HTTP 200, validating FR-12.

5.2 Acceptance Test

Manual acceptance testing was conducted to validate requirements that are difficult to automate reliably in a headless environment:

- **Concurrent editing (FR-5, FR-6)**: Two browser windows were opened on the same document, typed simultaneously, and the result was verified to converge without loss. Cursor colours and author labels appeared correctly.
- **AI assistant (FR-8)**: Selected text was submitted with various instructions; the rewritten output was inspected for coherence and correctness.
- **Admin dashboard (FR-10)**: The real-time presence indicator, usage-statistics charts, and user file-system tree were verified visually as users performed actions.
- **Notification delivery (FR-9)**: A document was shared between two accounts in separate browser sessions; the notification bell updated immediately in the recipient's session.
- **Load balancer behaviour**: With Docker Compose running, one backend container was stopped mid-session; the client reconnected automatically and continued editing.

Manual testing was unavoidable for these scenarios because they require a running browser with Tiptap rendering, two simultaneous sessions, or Docker orchestration that is impractical to reproduce inside a unit-test runner.

6 Deployment

6.1 Prerequisites

- Docker Engine $\geq$ 24.0 and Docker Compose $\geq$ 2.0.
- A Google Gemini API key (the free tier is sufficient for development).
- No other software is required on the host machine.

6.2 Configuration

A `.env` file in the project root provides all runtime configuration:

```
PORT=3000
MONGO_URI=mongodb://mongo:27017/dok
REDIS_URL=redis://redis:6379
JWT_SECRET=<your-secret-key-change-in-production>
GEMINI_API_KEY=<your-google-api-key>
```

The `JWT_SECRET` must be changed to a cryptographically random string in any non-development environment.

6.3 Docker Compose Architecture

The `docker-compose.yml` defines five services:

`nginx` Built from `nginx:alpine`; mounts `nginx.conf` read-only; exposes port 3000 to the host; depends on `backend1` and `backend2`.

`backend1, backend2` Built from `server/Dockerfile` (Node.js 22-Alpine); receive the environment variables from `.env`; depend on `mongo` and `redis`.

`mongo` `mongo:latest` image; port 27017 exposed to the host for development inspection.

`redis` `redis:alpine` image; port 6379 exposed to the host.

Server Dockerfile. The server image uses `node:22-alpine` as the base. Native addons for bcrypt require compilation; the Dockerfile installs build tools (`make`, `gcc`, `g++`, `musl-dev`, `linux-headers`) before running `npm install`. The container runs `npm run dev` (i.e., `tsx watch src/index.ts`), which provides hot-reload during development.

6.4 Starting the System

```
# Clone the repository
git clone <repository-url>
cd dok

# Create and populate the .env file (see above)

# Build and start all services
docker compose up --build

# The application is available at:
#   http://localhost:3000          (REST API + WebSocket, via Nginx)
#   http://localhost:3000/api-docs (Swagger UI)
```

Expected outcome. After approximately 30–60 seconds for image builds, all five containers start. MongoDB and Redis emit ready signals visible in the Docker Compose logs.

6.5 Frontend Development

The frontend is *not* included in the Docker Compose setup (by design, to keep build times short during development). It is run separately:

```
cd client
npm install
npm run dev
# SPA available at http://localhost:5173
```

The Vite dev server is pre-configured to proxy API requests to `http://localhost:3000`.

7 User Guide

7.1 Registration and Login

Navigate to `http://localhost:5173`. Click "Register" and provide a first name, last name, email address, and password. The first account created is automatically promoted to Administrator. Subsequent registrations create regular user accounts. After registration, you are redirected to the dashboard and a JWT is stored in the browser.

7.2 Dashboard

The main dashboard shows three panels:

- **My Doks**—private documents and folders owned by the current user.
- **Public Doks**—all public documents on the platform.
- **Shared with me**—documents that other users have shared with the current user.

Documents can be created, renamed, and deleted from this view. Clicking a document opens the collaborative editor.

7.3 Collaborative Editor

The editor is a full-featured rich-text environment powered by Tiptap. The toolbar provides: bold, italic, underline, strikethrough, highlight, text colour, font family, text alignment, headings, bullet and numbered lists, blockquotes, and undo/redo.

Active collaborators' cursors appear in real time, labelled with their name and a unique colour.

To share the document, use the "Share" button in the top-right corner; enter the recipient's email address and choose a role:

- **Viewer**: can read the document only.
- **Commenter**: can read and add/delete their own comments.
- **Editor**: full editing and commenting rights, but cannot delete the document or change sharing settings.

The AI assistant is accessed by selecting text and clicking the AI button in the bubble menu; type a natural-language instruction (e.g., "make this more concise") and confirm.

7.4 Notifications

The bell icon in the navigation bar shows the unread notification count. Notifications are delivered in real time when a document is shared with you or your permissions are changed.

7.5 Administrator Dashboard

Administrators access the admin panel from the navigation bar. Available views:

- **User list**: all registered users with online/offline status (live), registration date, and quick access to their details.
- **User detail**: per-user metrics (documents created, shared), file-system tree (private, public, shared documents), and access-log history.
- **Statistics**: platform-wide aggregated charts (users, documents, folders, sharing activity, site-access trends by hour/weekday/day-of-month).
- **Document logs**: real-time audit trail of access and modification events for any document.
- **Global access logs**: all login/logout events across all users.

8 Self-evaluation

8.1 Lorenzo Tosi

Role.

Strengths.

Weaknesses.

9 Future Works

Discuss possible future works, improvements, extensions, and optimizations.

MongoDB Replica Set. Deploy MongoDB as a three-node replica set (one primary, two secondaries) to provide read scalability and automatic failover at the database layer, removing the current single point of failure in the data tier.

JWT Refresh Tokens. The current implementation issues non-revocable access tokens valid for 24 hours. A refresh-token mechanism (stored in Redis with an invalidation list) would allow immediate revocation on logout or account compromise.

Document export. Since `tiptapJson` is already stored alongside the Yjs binary state, adding server-side PDF or DOCX export (e.g., via `pandoc` or a headless browser) would be straightforward.

Full Docker integration of the frontend. Adding the Vite build and an Nginx static-file server for the frontend to the Docker Compose stack would make the full application deployable with a single `docker compose up` command.

References