# Pixie
# Collaborative Pixel Art Platform

### Final Report for the Distributed Systems Course 2025/2026

Andrea Zavatta
andrea.zavatta3@studio.unibo.it

Matteo Susca
matteo.susca@studio.unibo.it

June 14, 2026

Pixie is a collaborative platform for creating pixel art in real time. Multiple users interact simultaneously on the same digital canvas, cooperating on the creation of a shared drawing through an interactive web interface. The application is designed as a Single Page Application (SPA) whose central element is a grid of pixels whose state is kept consistent and updated for all connected clients, enabling a uniform view of the drawing in progress.

The system follows a three-tier architecture comprising a Vue.js frontend, a Node.js/Express backend, and a MongoDB database. Real-time synchronisation relies on Socket.IO for bidirectional communication and Server-Sent Events (SSE) for push notifications. To support horizontal scalability and fault tolerance in a distributed deployment, the architecture integrates Redis as a shared state store and pub/sub message broker, Traefik as a load-balancing reverse proxy, and a coordination service that uses distributed locking to safely persist canvas data from any server instance.

## 1 Concept

Pixie is a web-based application designed as a Single Page Application (SPA), providing a graphical interface through which multiple users can interact with a shared digital canvas in real time to create pixel art together. The system follows a three-tier architecture:

- **Presentation tier**: the SPA running in the user's browser, responsible for rendering the canvas, capturing user input, and communicating asynchronously with the backend.

- **Application tier**: one or more stateless server instances that process requests, handle real-time bidirectional communication, and run the application logic.
- **Data tier**: a persistent database for durable storage of domain entities, paired with a distributed in-memory store for fast state management and inter-instance coordination.

### 1.1 Usage Scenario

Users join dedicated "lobbies' each hosting a shared canvas where every pixel drawn by one participant is synchronised instantly to all others. Users are geographically distributed and connect via desktop or mobile web browsers. During a drawing session they generate a continuous stream of events—drawing, zooming, panning—through clicks or continuous dragging on an HTML5 canvas element, with sessions lasting from minutes to hours.

The system persists user profiles, notification history, and lobby metadata in a durable database, while the live canvas state is kept in a distributed in-memory store for fast access and cross-instance synchronisation. Canvas data is periodically flushed to the database for long-term persistence. The application distinguishes three user roles:

- *Base User*: can draw, join lobbies, view connected users, and export canvases.
- *Creator*: the owner of a specific lobby, with moderation powers (kick, ban, clear canvas, delete lobby).
- *Administrator*: holds global privileges to manage all system content and users.

### 1.2 Motivation for Distribution

Distribution is a fundamental requirement for Pixie. A shared pixel grid messaging and state-management layer for consistency, while low-latency propagation of drawing updates requires in-memory state management rather than direct database access on every operation. The architecture is designed to scale horizontally behind a load balancer, and distributed caching with a write-behind persistence strategy and distributed locking makes the system resilient to individual node failures, bounding potential data loss to at most one flush interval.

## 2 Requirements Elicitation and Analysis

### 2.1 Functional Requirements

- **FR1 – User Access and Profile**: the system allows access via username without a password. Upon first login with an unknown username, a new user record is automatically created. Users can modify their username from the profile page.
  - **AC1**: a new user record is successfully created in the database upon first login with a unique username.

  - **AC2**: changes to a username in the profile view are persisted and immediately visible across the application.

- **FR2 – Lobby Management**: authenticated users can create lobbies specifying name, description, canvas dimensions, maximum capacity, and colour palette. They can list available lobbies with filters. Owners can delete their own lobbies.
  - **AC1**: lobbies are created with valid constraints and appear correctly in the filtered search list.
  - **AC2**: deleting a lobby disconnects all active participants and removes the associated data from the system.

- **FR3 – Collaborative Drawing and Tools**: users can draw on the canvas via single clicks or continuous dragging with real-time propagation. Colours are selected from a palette defined at lobby creation.
  - **AC1**: drawing updates are synchronised to all lobby participants in real time.
  - **AC2**: the "Clear Canvas" action resets the grid for all users, but only when triggered by an authorized owner or admin.

- **FR4 – Room Awareness and Moderation**: the system provides a real-time list of connected users and empowers lobby owners to manage participants through kick or ban actions.
  - **AC1**: the user list updates dynamically as participants join or leave a room.
  - **AC2**: banned users are immediately disconnected and prevented from re-entering the lobby.

- **FR5 – Notification System**: the software supports persistent, real-time push notifications, allowing users to view their history and mark items as read.
  - **AC1**: push notifications are delivered in real time and stored for later retrieval if the user is offline.
  - **AC2**: marking a notification as "read" updates the unread count in the UI in real time.

- **FR6 – Canvas Export**: users can export the current state of the canvas as a PNG image.
  - **AC1**: the system generates a valid PNG file representing the current pixel grid state upon user request.

## 2.2 Non-Functional Requirements

- **NFR1 – Performance (Latency and Rendering)**: propagation of drawing events must maintain low latency. The canvas must be rendered efficiently to prevent UI freezing.

  - **AC1**: end-to-end latency for pixel updates remains consistently low under normal network conditions (design target: below 100 ms).
  - **AC2**: the UI maintains smooth rendering during zoom and pan operations by utilising off-screen canvas rendering (design target: 60 fps).

- **NFR2 – Reliability**: canvas state must be periodically persisted to prevent data loss. In case of duplicate sessions from the same user, the most recent connection must prevail.
  - **AC1**: the write-behind cache successfully persists canvas data to the database at regular intervals.
  - **AC2**: establishing a new session automatically terminates any existing duplicate connections for that user.

- **NFR3 – Security**: all APIs and real-time connections require mandatory JWT authentication; the system must also validate permissions and configure CORS correctly.
  - **AC1**: unauthorized requests without a valid token are rejected with a 401 error.
  - **AC2**: the system strictly validates that only authorized roles can perform moderation tasks.

- **NFR4 – Scalability**: the system must support horizontal scaling across multiple server instances with a shared coordination layer.
  - **AC1**: real-time events are successfully synchronised across all server instances.
  - **AC2**: memory usage is reclaimed by automatically offloading inactive lobbies from the in-memory store.

## 2.3 Implementation Requirements

- **IR1 – Tech Stack**: the system is built using the MEVN stack (MongoDB, Express, Vue, Node) and TypeScript.

## 2.4 Relevant Distributed System Features

**Transparency.** Pixie provides *location transparency* (users interact with a single URL, unaware of which backend instance handles their requests) and *replication transparency* (a shared in-memory store presents a single logical state to all instances). *Failure transparency* is partial: the load balancer reroutes traffic if an instance crashes, but affected users experience a brief disconnection before automatic reconnection.

**Fault tolerance.** Canvas state is periodically flushed to the database via a write-behind strategy coordinated by a distributed lock, bounding data loss to one flush interval. Graceful shutdown guarantees a final flush (see section 3.7).

**Scalability.** Multiple backend instances run behind a load balancer sharing the same external services. A real-time adapter broadcasts events across instances, and atomic counters enforce lobby capacity limits (see section 3.8).

**Security.** JWT authentication is required on every REST, WebSocket, and SSE connection. Authorization follows RBAC with three roles. CORS restricts API access to trusted origins (see section 3.9).

**Resource sharing.** The canvas is the primary shared resource: concurrent pixel writes use atomic byte-offset operations, dirty-tracking identifies canvases needing persistence, and a pub/sub channel distributes notifications across instances.

**Openness and maintainability.** The system exposes a RESTful API (OpenAPI 3.1.0, Swagger UI), uses standard protocols (HTTP, WebSocket, SSE) and formats (JSON, binary buffers), and is fully containerised. The codebase follows Separation of Concerns with a layered backend and MVVM frontend, both in TypeScript.

**Performance and economy.** Sub-millisecond Redis writes at specific byte offsets, batched and redundancy-filtered pipelines, and off-screen canvas rendering keep latency low. Pixel data stored as 1-byte palette indices reduces bandwidth $\sim 7\times$; idle lobbies are automatically offloaded from memory.

## 2.5 Non-Relevant Distributed System Features

**Strict consistency.** Strict data consistency across all nodes and storage tiers is not a requirement for this project. As a collaborative drawing application, low-latency propagation of user inputs is prioritised over absolute transaction correctness. The system explicitly favours availability (AP in the CAP theorem), accepting minor staleness in the durable MongoDB snapshot and resolving concurrent pixel writes on the same byte offset with a simple last-writer-wins semantic.

**Heterogeneity.** Heterogeneity of server components is not relevant to this architecture. The application tier is designed to be scaled horizontally, but it consists of entirely homogeneous, identical Node.js replicas running the exact same codebase. There is no requirement to integrate legacy systems or coordinate among distributed microservices written in different programming languages.

**Economy and strict operating costs.** While memory usage and network bandwidth were minimised to ensure good performance (e.g., using 1-byte palette indices), strict economy and long-term cloud operating costs were not primary architectural drivers. The system is designed as an academic prototype to be easily self-hosted via Docker Compose on standard hardware, rather than being heavily optimised for specific cloud provider billing models or strict budget constraints.

# 3 Design

## 3.1 Architecture

Pixie adopts a **three-tier client–server architecture** with a **layered internal structure** for each tier, chosen for separation of concerns, horizontal scalability, and real-time communication support. The stateless application tier—enabled by externalising shared state to a distributed cache—allows multiple backend instances to run behind a load balancer without code changes. The architecture supports three complementary communication channels:

- *REST API*: for standard CRUD operations (lobby management, user profiles, authentication, moderation).
- *WebSocket* (via Socket.IO): for bidirectional, high-frequency real-time communication (canvas synchronisation, room awareness).
- *Server-Sent Events (SSE)*: for unidirectional push notifications from server to client.

### 3.1.1 Backend Layered Architecture

The backend implements a layered architecture that separates responsibilities into well-defined strata:

- **Presentation Layer** (Routes): defines API endpoints and composes authentication/authorization middlewares.
- **Application Layer** (Controllers): handles HTTP requests, validates input, and orchestrates operations by delegating to services.
- **Business Logic Layer** (Services): encapsulates domain rules, complex operations, and coordinates data access.
- **Data Access Layer** (Models): defines data schemas, validation constraints, and the interface with the database.

Cross-cutting concerns (JWT authentication, role-based authorization, and centralised error handling) are implemented as middleware applied at the route level before controller execution.

### 3.1.2 Frontend MVVM Architecture

The frontend follows the Model–View–ViewModel (MVVM) pattern:

- **Model**: Pinia stores maintain the application state (user profile, lobby data, canvas pixels, notifications).
- **View**: Vue.js templates render the user interface.

- **ViewModel**: the Composition API (`ref`, `computed`, composables) exposes reactive properties and methods that bind the model to the view.

Components follow the *props-in / events-out* pattern and are organised by domain. Reusable logic is extracted into composables (e.g. the pixel-canvas composables for transformations, rendering, and input handling).

## 3.2 Infrastructure

The system is composed of the following infrastructural components:

- **Clients** (browsers): Vue.js Single Page Applications running in the user's web browser. They communicate with the backend through the reverse proxy.
- **Reverse proxy / load balancer** (Traefik): the single entry point for all external traffic. It routes requests based on URL path: `/api/*` and `/socket.io/*` are forwarded to the backend server pool, while all other paths are served by the frontend container. In production, Traefik also performs load balancing across multiple backend replicas.
- **Backend servers** (Node.js / Express): one or more stateless application server instances. Each instance connects to the shared Redis and MongoDB services. Instances are not directly exposed to the network; they are reachable only through Traefik within the Docker network.
- **Frontend server** (Nginx): serves the compiled static assets of the Vue.js SPA. In production, the client container runs Nginx and is registered with Traefik for the catch-all route.
- **Distributed cache / message broker** (Redis): serves three roles: (1) shared in-memory store for canvas pixel data and lobby metadata, (2) pub/sub message broker for the Socket.IO adapter and SSE notifications, and (3) coordination primitive for distributed locking and atomic counters.
- **Database** (MongoDB): provides durable persistence for user profiles, lobby definitions, canvas snapshots, and notification history. It is not exposed outside the Docker network.

### 3.2.1 Network Distribution

In the default deployment all components reside on a single Docker host. However, Redis and MongoDB can be hosted externally (via `REDIS_URL` and `MONGO_URI`), and multiple backend containers can run on different machines as long as they share the same endpoints.

### 3.2.2 Service Discovery and Naming

Components discover each other through Docker's built-in DNS resolution within the `pixie-net` bridge network. Service names defined in `docker-compose.yml` (e.g. `mongo`, `redis`, `server`) are resolved to container IP addresses automatically.

Traefik discovers backend instances via the Docker socket API (`/var/run/docker.sock`) and Docker labels. This design was chosen deliberately: when a new backend container is added (e.g. via `docker compose up -d --scale server=N`), Traefik detects it automatically through the Docker event stream and includes it in the load-balancing pool. No manual registration, configuration reload, or restart is needed—the system adapts to topology changes at runtime.

### 3.2.3 Sticky Sessions

WebSocket connections are stateful: once established, all frames must be delivered to the same backend instance. Traefik is configured with sticky sessions using a cookie (`PIXIE_SESSION`), ensuring that the initial HTTP upgrade request and all subsequent WebSocket frames from a given client are routed to the same backend. The same cookie also ensures that SSE connections and REST API calls from the same browser session are served by the same instance, which is beneficial for locality (e.g. the SSE client list is per-instance).

## 3.3 Modelling

### 3.3.1 Domain Model

The system defines four core domain entities:

- **User**: a registered user with a unique username, an admin flag, and timestamps. Users are persisted in MongoDB and queried through the backend's data access layer.
- **Lobby**: a named collaborative room with a maximum capacity, an optional description, a reference to its owner (User), a reference to its Canvas, and a list of banned users. Lobbies are persisted in MongoDB. The current occupant count is tracked as a Redis atomic counter to allow cross-instance increment/decrement.
- **Canvas**: the pixel grid associated with a Lobby, storing width, height, a colour palette (array of hex strings, max 256 entries), and pixel data as a binary buffer (one byte per pixel, indexing into the palette). Canvas data has a dual residency: actively used canvases (*hot data*) live in Redis as a binary string with metadata in a hash for sub-millisecond access, while durable snapshots (*cold data*) are stored in MongoDB and updated periodically by the write-behind flush worker.
- **Notification**: a message sent to a specific user, comprising a title, body, read status, and timestamps. Persisted in MongoDB.

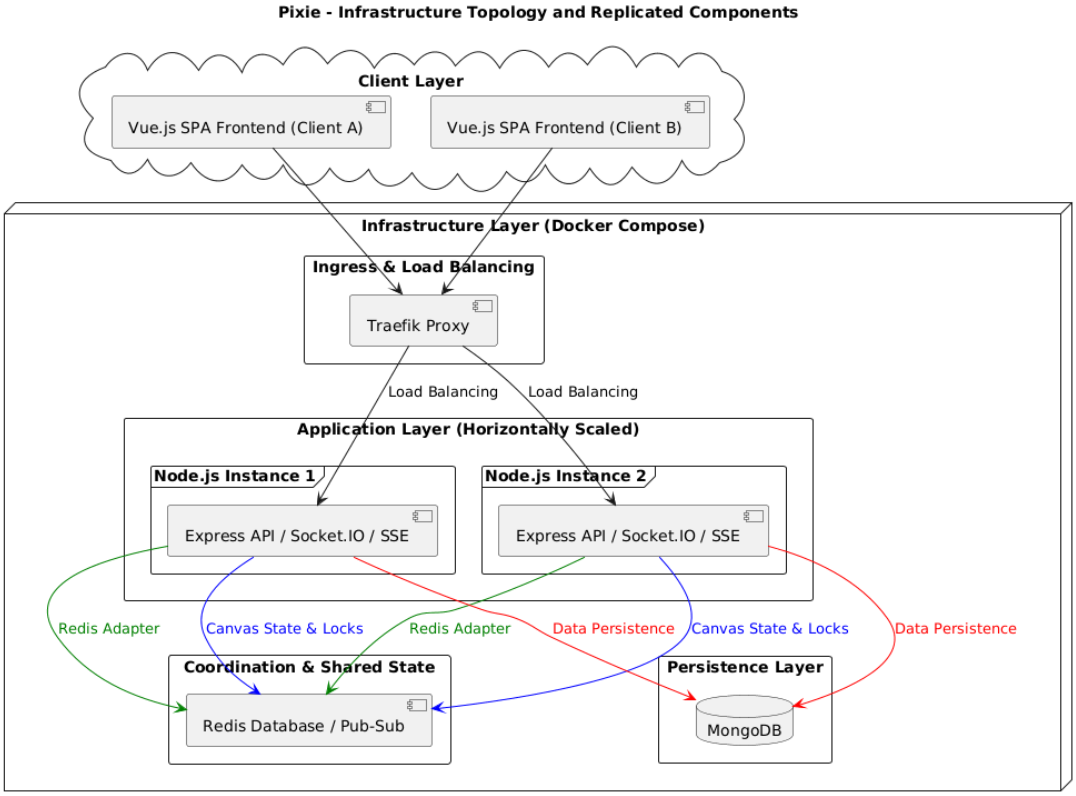


Figure 1: Architecture topology

Client-side state (selected colour, zoom level, pan position, pending draw buffer) is ephemeral and maintained exclusively in the browser's stores and composable state.

### 3.3.2 Events, Messages, and System State

The main domain events that drive state changes are:

- *User registered* (first login with a new username)
- *Lobby created* / *Lobby deleted*
- *User joined lobby* / *User left lobby*
- *Pixel drawn* (single) / *Pixels drawn* (batch) — canvas clearing is a special case where all pixels are set to the default colour (index 0)
- *User kicked* / *User banned* / *User unbanned*
- *Notification sent* / *Notification read*

These events are carried across three communication channels:

- **Client-initiated requests**: sent via WebSocket (`JOIN_LOBBY`, `DRAW`, `DRAW_BATCH`, `CLEAR_CANVAS`) or REST API calls (create/delete lobby, kick, ban, profile update).
- **Server-broadcast events**: propagate state changes to all clients in a lobby — `INIT_STATE` (full canvas on join), `PIXEL_UPDATE` / `PIXEL_UPDATE_BATCH` (drawing), `CANVAS_CLEARED`, `USER_JOINED` / `USER_LEFT`, `FORCE_DISCONNECT`, `LOBBY_DELETED`, and `BANNED_USERS_UPDATED`.
- **Push notifications**: delivered via SSE for asynchronous, unidirectional events (e.g. ban, notifications).

The global system state comprises users, lobbies, and notifications (all in MongoDB), live pixel data of active canvases (Redis, periodically synced to MongoDB), and transient WebSocket/SSE connection sets (in memory). Redis holds the following key structures for each active lobby:

- Hash `lobby:{id}:meta`: canvas dimensions and palette.
- String `lobby:{id}:canvas`: binary pixel data.
- String `lobby:{id}:count`: atomic occupant counter.
- Set `lobbies:dirty`: IDs of canvases with unflushed changes.
- String `pixie:db_save_lock`: distributed lock for flush coordination.
- Pub/sub channels: `notifications` and Socket.IO adapter channels.

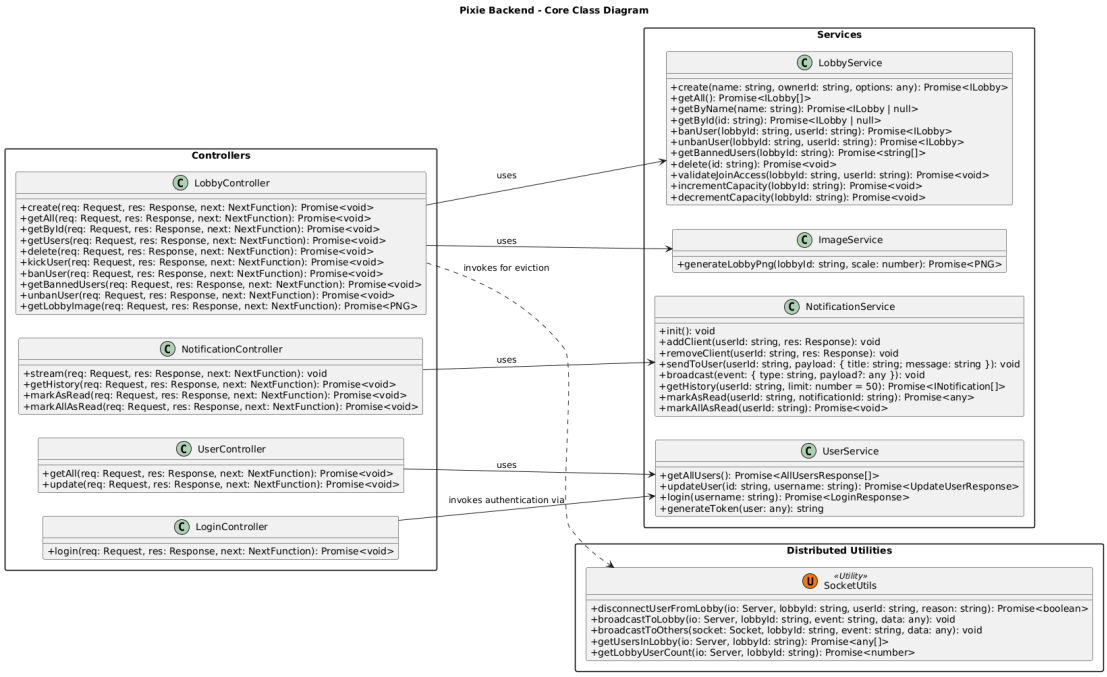


Figure 2: Class diagram

## 3.4 Interaction

Components communicate through three distinct channels, each chosen to match the interaction pattern's requirements:

### 3.4.1 REST API Interactions

Standard request–response interactions over HTTP. Used for all CRUD operations:

- `GET /api/lobbies`: client fetches the list of available lobbies.
- `GET /api/lobbies/:id/image`: client requests a PNG export; server generates a PNG of the current canvas state and streams it.
- `POST /api/login`: client sends a username; server creates the user if new, generates a JWT, and returns it.
- `POST /api/lobbies`: client creates a new lobby with specified parameters.
- `POST /api/lobbies/:id/kick` and `/ban`: owner triggers moderation; server disconnects the target user and (for bans) persists the ban list and sends a notification.
- `DELETE /api/lobbies/:id`: owner requests lobby deletion; server disconnects all sockets in the room and deletes the data.

### 3.4.2 WebSocket Interactions (Socket.IO)

Bidirectional, event-driven communication for real-time operations. The interaction follows a *command–broadcast* pattern:

1. Client emits a command (e.g. `DRAW_BATCH`).
2. Server validates the command (checks room membership, permissions).
3. Server applies the state change to the shared store.
4. Server broadcasts the result to all clients in the room (e.g. `PIXEL_UPDATE_BATCH`).

The `JOIN_LOBBY` flow is more complex: it involves ban verification, duplicate session handling (last-wins policy), atomic capacity checking, room join, user list broadcast, and initial state transfer.

### 3.4.3 SSE Interactions

Unidirectional server-to-client push. A long-lived HTTP connection is established at `GET /api/notifications/stream`. The server sends events (e.g. ban notifications, lobby update signals) as they occur. A heartbeat comment is sent every 30 seconds to prevent proxy timeouts.

When a notification must reach a user who may be connected to any backend instance, the originating instance publishes to a Redis pub/sub channel. All instances receive the message and deliver it to any local SSE clients matching the target user ID.

## 3.5 Behaviour

### 3.5.1 Startup and Initialisation

When a backend instance starts, it connects to MongoDB and Redis, creates the Socket.IO server with the Redis Adapter (for cross-instance broadcasting), subscribes to the Redis `notifications` pub/sub channel, starts the periodic flush worker, and begins listening for HTTP and WebSocket connections. Each instance is fully independent and identical—no instance is designated as primary.

### 3.5.2 Drawing Flow

When a user draws on the canvas, the client-side store accumulates pixel changes in a local buffer using Bresenham's line algorithm for stroke interpolation. At stroke end, a single `DRAW_BATCH` event is emitted to the server.

The server validates room membership and permissions, then writes each changed pixel to Redis via atomic byte-offset operations. Redundant writes (where the pixel already holds the requested colour) are filtered out. The lobby is added to the `lobbies:dirty` set, and the resulting updates are broadcast to all clients in the room. The Redis Adapter transparently replicates the broadcast to clients connected to other backend instances.

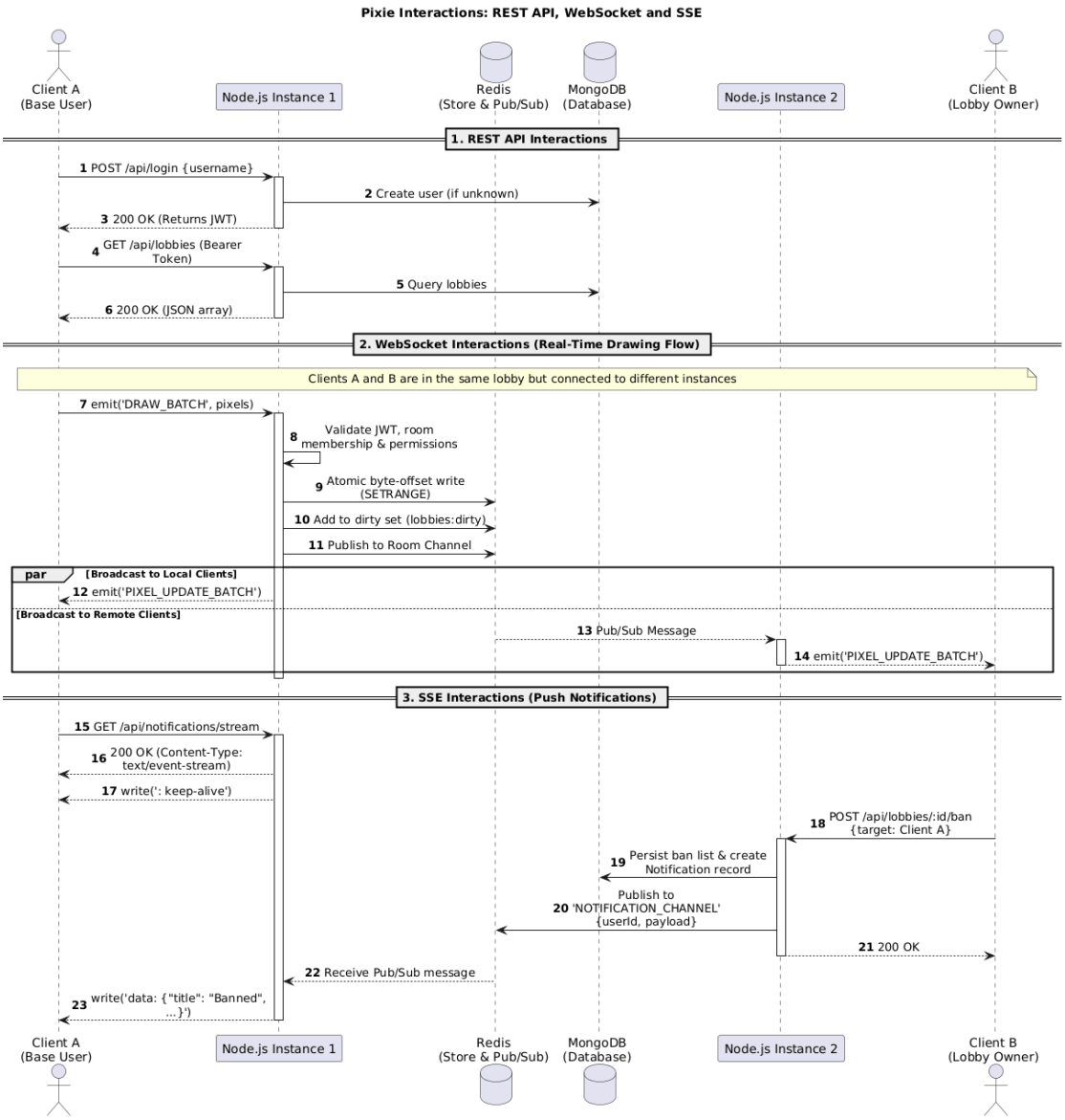


Figure 3: Interaction diagram

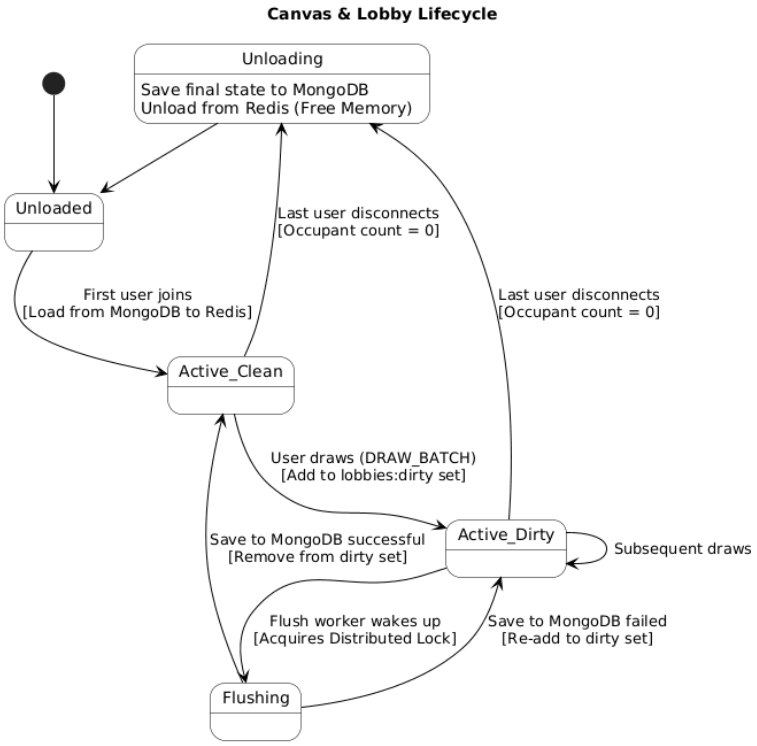


Figure 4: State diagram

On receiving the broadcast, each client applies the pixel changes directly to its local buffer and triggers a re-render of the off-screen canvas.

### 3.5.3 Periodic Flush (Write-Behind Persistence)

Every few seconds, each backend instance's flush worker attempts to acquire the distributed lock. The instance that succeeds reads the `lobbies:dirty` set, persists each dirty canvas to MongoDB, and releases the lock. Only one instance flushes at a time, preventing duplicate writes. If a save fails, the lobby is re-added to the dirty set for retry.

### 3.5.4 Disconnection and Cleanup

When a client disconnects (voluntarily, due to a crash, or because of a failed heartbeat), the server decrements the lobby's occupant counter and broadcasts `USER_LEFT` to the room. If the lobby becomes empty after disconnection, its canvas data is saved to MongoDB and then unloaded from Redis, freeing memory.

On graceful shutdown (SIGINT/SIGTERM), the backend instance stops accepting new connections, performs a final flush of all dirty lobbies, closes its Redis and MongoDB connections, and exits. A 10-second deadline ensures termination even if the flush stalls.

## 3.6 Data and Consistency Issues

### 3.6.1 Storage and Access Patterns

All four domain entities (User, Lobby, Canvas, Notification) are stored in MongoDB as documents.

- **Backend services** are the only components that query the database directly. Clients never access MongoDB; all data is mediated by the REST API or WebSocket events.
- **Concurrent reads** occur when multiple users request lobby listings or when the flush worker reads lobby references to save canvas data. These are safe because MongoDB reads are consistent within a single document.
- **Concurrent writes** are minimised by design: user profile updates and lobby CRUD are low-frequency. The only high-frequency write target—canvas pixel data—is buffered in Redis and flushed by a single writer (the instance holding the distributed lock) at a time.

### 3.6.2 Shared Data and Coordination

The canvas pixel data is the primary piece of shared mutable state. It must be accessible to all backend instances for reads (when a user joins a lobby) and writes (when a user draws). Redis serves as the shared medium, with the following coordination mechanisms:

- **Pixel writes**: each pixel is written via an atomic byte-offset operation. Concurrent writes to different pixels are naturally serialised. Concurrent writes to the *same* pixel follow a last-writer-wins semantic, which is acceptable for a collaborative drawing application.
- **Lobby capacity**: the occupant count is maintained as an atomic counter, preventing race conditions when multiple users attempt to join the same lobby simultaneously from different backend instances.
- **Dirty tracking**: a Redis set is updated atomically whenever a pixel is written. The flush worker reads the set and removes entries before saving. If a new write occurs during flushing, the lobby is re-added, ensuring no changes are lost.

### 3.6.3 Consistency Model

The system favours *availability* and *low latency* over strict consistency. The canvas state in Redis is the authoritative real-time state; MongoDB contains a slightly stale snapshot (at most one flush interval behind). This is an explicit trade-off: for a collaborative drawing application, a few seconds of potential data loss in a catastrophic failure is acceptable, while low latency for drawing updates is essential for usability.

## 3.7 Fault-Tolerance

### 3.7.1 Data Replication and Write-Behind Caching

Canvas data is replicated across two storage tiers: **Redis** (primary, fast) for live pixel data, and **MongoDB** (secondary, durable) for periodic snapshots. A flush worker on every backend instance runs every 5 seconds: it attempts to acquire the distributed lock (`pixie:db_save_lock`) via a set-if-not-exists operation with a 10-second TTL. The winning instance reads the dirty set, persists each canvas to MongoDB, and releases the lock via an atomic compare-and-delete Lua script. Failed saves are retried. This ensures exactly one instance flushes at a time. See section 4 for the lock implementation details.

### 3.7.2 Heartbeating and Timeouts

- **SSE keep-alive**: a comment is sent every 30 seconds on each SSE connection to prevent intermediate proxies from closing idle connections.
- **Socket.IO heartbeat**: Socket.IO has a built-in ping/pong mechanism (default interval: 25 s, timeout: 20 s). If a client fails to respond, the server closes the connection and triggers cleanup (broadcasts `USER_LEFT`, decrements the lobby capacity counter).
- **Distributed lock TTL**: the flush lock has a 10-second TTL. If an instance crashes while holding the lock, Redis automatically expires it, allowing another instance to take over within at most one TTL period.
- **Graceful shutdown timeout**: a 10-second deadline ensures that the final flush and cleanup complete before the process is forcefully terminated.

### 3.7.3 Error Handling

- **Backend crash**: Traefik stops routing traffic to the failed instance. Users on that instance lose their WebSocket connection; the Socket.IO client library attempts automatic reconnection, which is routed to a surviving instance. Canvas data in Redis is unaffected.
- **Redis failure**: canvas operations fail gracefully (drawing stops, joins fail), but the application does not crash. The last MongoDB snapshot remains available for recovery.
- **MongoDB failure**: on startup, the server exits if MongoDB is unreachable. During operation, individual query failures are caught and returned as HTTP errors; the flush worker logs failures and retries.
- **REST API errors**: a global Express error handler catches all unhandled errors and returns structured JSON responses. On the client side, an Axios interceptor catches HTTP errors and displays them via the toast notification system.

## 3.8 Availability

### 3.8.1 Caching

The Redis layer functions as a *write-behind cache* for canvas data, sitting between the backend application servers and MongoDB. Direct database access for every pixel write would introduce unacceptable latency and overwhelm the database under high write loads. Redis provides sub-millisecond read and write operations on the canvas binary data.

On first access, canvas data is loaded from MongoDB into Redis using a request-coalescing pattern to prevent thundering-herd queries (see section 4). All subsequent reads and writes operate on Redis. Modified canvases are tracked in a dirty set and periodically flushed to MongoDB. Idle lobbies are unloaded from Redis after saving, freeing memory.

### 3.8.2 Load Balancing

Traefik sits in front of all backend server instances to distribute incoming requests and enable horizontal scaling. It uses the **Power-of-Two-Choices (P2C)** algorithm: for each new connection, Traefik randomly selects two backend instances and routes the request to the one with fewer active connections. Compared to a simple Least-Connections strategy, P2C achieves near-optimal load distribution without requiring centralised, globally consistent connection-count state—each routing decision is made with only two samples, making it more suitable for a dynamic, horizontally-scaled deployment. With a small replica count (2–5 instances), the practical difference is minor, but P2C scales better as the pool grows.

Sticky sessions (via the `PIXIE_SESSION` cookie) ensure that all requests from the same browser session are routed to the same backend, which is required for WebSocket connection affinity and beneficial for SSE connection locality.

### 3.8.3 Network Partitioning Behaviour

The system prioritises *availability* over strict *consistency* in the event of a network partition (AP in the CAP theorem). During a partition, each reachable subset of the system continues to operate independently:

- **Client–server partition**: if a client loses connectivity, the Socket.IO client detects the disconnection and attempts automatic reconnection. Drawing operations performed locally during the partition are not persisted. On reconnection, the client re-joins the lobby and receives the full canvas state (`INIT_STATE`), converging to the authoritative server-side state.
- **Backend–Redis partition**: affected instances cannot perform canvas operations or cross-instance broadcasts. They continue serving static content and non-canvas REST endpoints. Once connectivity is restored, normal operation resumes.

- **Backend–MongoDB partition**: active drawing sessions continue unaffected (canvas data lives in Redis). Operations requiring database writes (lobby creation, user registration) fail until connectivity is restored. The flush worker accumulates dirty entries and persists them once MongoDB becomes reachable again—ensuring eventual convergence.

In all cases, the system favours availability; convergence is restored once connectivity resumes.

## 3.9 Security

### 3.9.1 Authentication

Every interaction with the system requires a valid JSON Web Token (JWT):

- **REST API**: the client includes the token in the `Authorization: Bearer <token>` header. An Express middleware verifies the signature and expiry, attaching the decoded user object to the request.
- **WebSocket**: the token is sent during the Socket.IO handshake. A middleware verifies it before granting the connection. Invalid or expired tokens cause the connection to be rejected.
- **SSE**: the notification stream endpoint requires the same Bearer token as REST endpoints.

Tokens are generated with a 7-day expiry. On the client side, tokens are stored in `localStorage`. This is a known security trade-off: `localStorage` is accessible to any JavaScript running on the page, making it vulnerable to XSS attacks. The approach was chosen for simplicity; a more robust alternative would use an HttpOnly cookie (immune to XSS) with a short-lived access token (e.g. 15 minutes) refreshed against the cookie.

Expired tokens trigger an automatic logout via the Axios response interceptor (which detects 401 responses).

### 3.9.2 Authorization

The system implements Role-Based Access Control (RBAC) with three levels:

- **Base User**: can draw, join lobbies, view connected users, export canvases, and manage their own profile and notifications.
- **Lobby Owner** (Creator): inherits base user permissions within their lobby and additionally can kick/ban users, clear the canvas, manage the ban list, and delete the lobby.
- **System Administrator**: inherits all permissions globally. Administrators can manage any lobby (override owner checks), kick any user, and clear any canvas.

On the WebSocket side, event-level authorization is enforced: `DRAW` and `DRAW_BATCH` are permitted only if the sender is a member of the target room; `CLEAR_CANVAS` additionally checks that the sender is the lobby owner or an admin.

### 3.9.3 CORS and Network Isolation

The backend configures CORS to accept requests only from trusted origins. MongoDB and Redis are not exposed outside the Docker network—only Traefik is bound to host ports, so all database access is mediated by the application layer. The Nginx frontend container also sets `X-Frame-Options: SAMEORIGIN` and `X-Content-Type-Options: nosniff` to mitigate clickjacking and MIME-sniffing attacks.

# 4 Implementation

## 4.1 Technology Stack

- **Frontend**: Vue.js 3 with Composition API, Pinia for state management, Vue Router for navigation, Vite as the build tool. All code is written in TypeScript. The production build is served by Nginx.
- **Backend**: Node.js 22 with Express.js, Socket.IO for WebSocket management. All code is written in TypeScript and compiled to ESM.
- **Database**: MongoDB with Mongoose ODM for schema validation and query building.
- **Distributed infrastructure**: Redis (data store, pub/sub, locks), `@socket.io/-redis-adapter` for multi-instance WebSocket broadcasting, Traefik as reverse proxy and load balancer.
- **Testing**: Vitest as test runner (unit and integration), `mongodb-memory-server` for in-memory MongoDB during tests, Docker for infrastructure tests.
- **Containerisation**: Docker with multi-stage builds (dev/build/production stages), Docker Compose for orchestration.
- **API documentation**: OpenAPI 3.1.0 specification served via Swagger UI at `/api-docs`.

## 4.2 General Implementation Choices

Communication with the backend uses JSON for REST request and response bodies and for Socket.IO event payloads. The main exception is canvas pixel data: the server transmits the full canvas state as a raw `Uint8Array` of palette indices, which the client writes directly into an `ImageData` buffer (see below). SSE messages use the standard `text/event-stream` format.

Authentication relies on the `jsonwebtoken` library. On login, the server signs a token containing the user ID, username, and admin flag with a 7-day expiry. Every subsequent REST request must include the token in the `Authorization: Bearer` header; an Express middleware (`authenticateToken`) verifies the signature and attaches the decoded user to the request object. The same verification is performed during the Socket.IO handshake and on SSE connections.

Authorization is enforced through composable Express middlewares applied at the route level. `requireLobbyOwner` checks that the requesting user owns the target lobby (or is a system administrator); `requireLobbyAccess` verifies that the user is not banned. Routes compose these middlewares declaratively — for example, the delete-lobby route chains `authenticateToken` and `requireLobbyOwner` before the controller is reached.

## 4.3 Noteworthy Implementation Details

### 4.3.1 Two-Canvas Rendering Architecture

The canvas rendering system uses a *two-canvas architecture* that decouples pixel data management from visual display, achieving efficient rendering at arbitrary zoom levels.

The first canvas is an **off-screen pixel buffer**—a programmatically created `<canvas>` element that is never added to the DOM. It holds the pixel-accurate representation of the drawing at 1:1 scale (one CSS pixel per grid pixel). The composable `usePixelBuffer` manages this buffer: when the full canvas state arrives from the server (as a `Uint8Array` of palette indices), it converts each index into a packed ABGR integer via a pre-computed `Uint32Array` lookup table and writes the result directly into the `ImageData` backing buffer:

```
const data32 = new Uint32Array(imageData.data.buffer);
for (let i = 0; i < len; i++) {
  data32[i] = palette32[pixels[i]];
}
pixelCtx.putImageData(imageData, 0, 0);
```

This avoids per-pixel `fillRect` calls and string-to-colour parsing, achieving $O(n)$ buffer fills with minimal garbage collection pressure. The lookup table and `ImageData` objects are cached and only rebuilt when the canvas dimensions or palette change. For incremental updates (single-pixel changes from other users), a direct `fillRect` call bypasses the full-buffer path, keeping incremental renders sub-millisecond.

The second canvas is the **visible display canvas** in the DOM. The composable `useCanvasRenderer` redraws it on every frame that requires a visual update (zoom, pan, or pixel change) by calling:

```
ctx.imageSmoothingEnabled = false;
ctx.drawImage(
  pixelBuffer,
  0, 0, canvasWidth, canvasHeight,
  panX, panY, width * scale, height * scale
);
```

The `drawImage` call scales the off-screen buffer onto the display canvas using the current zoom factor and pan offsets, without re-computing any pixel data. Setting `imageSmoothingEnabled = false` preserves the sharp, blocky aesthetic expected of pixel art. When the zoom level exceeds 4×, a grid overlay is drawn on top to delineate individual pixels.

This separation means that zoom and pan operations—the most frequent user interactions—only involve a single `drawImage` call on already-rendered data, regardless of canvas size.

### 4.3.2 Double-Pipeline Batch Pixel Writes with Redundancy Filtering

The `CanvasStore.modifyPixelBatch` method implements a *two-pipeline* Redis transaction strategy to handle batch drawing operations efficiently. The first pipeline reads the current colour of every target pixel via `GETRANGE`. The results are compared against the requested colours to filter out redundant writes (where the pixel already holds the requested value). Only the pixels that actually change are written in a second pipeline using `SETRANGE`, all within a single `MULTI/EXEC` transaction. This design minimises unnecessary Redis writes and network round-trips: a stroke of 50 pixels that overlaps with existing content may result in only 10 actual writes.

### 4.3.3 Distributed Lock with Lua-Based Safe Release

The `CoordinationService` uses Redis `SET NX PX` for lock acquisition with a unique per-instance owner value (`SERVER_ID`). The critical design choice is in the lock release: a Lua script atomically checks that the lock is still owned by the releasing instance before deleting it:

```
if redis.call("get", KEYS[1]) == ARGV[1] then
  return redis.call("del", KEYS[1])
else
  return 0
end
```

This compare-and-delete pattern prevents a dangerous race condition where Instance A's lock expires due to a slow flush, Instance B acquires it, and then Instance A's release call erroneously deletes Instance B's lock. The Lua script executes atomically within Redis, eliminating the race window entirely.

### 4.3.4 Request Coalescing for State Hydration

The `CanvasService.getState` method implements a *request coalescing* (single-flight) pattern—described in the caching design in section 3.8—at the implementation level. A `pendingLoads` Map stores the Promise of the first database query for a given lobby; any concurrent calls for the same lobby reuse that Promise. The map entry is cleaned up in the `finally` block, ensuring correctness regardless of success or failure.

### 4.3.5 Decoupled Real-Time Notifications via SSE

To support horizontal scaling, we implemented a real-time notification system using Server-Sent Events (SSE) orchestrated via Redis Pub/Sub. In a distributed environment, a user's SSE connection is tied to a specific backend replica; therefore, we decoupled event origination from delivery.

When a system event occurs, the originating instance publishes the notification to a shared Redis channel. All backend instances subscribe to this channel, checking their local memory for active connections matching the target user ID. This "broadcast-to-all, filter-at-edge" pattern ensures reliability across replicas without requiring a centralized session store. To maintain connection longevity through proxies like Traefik, we inject a 30-second heartbeat:

```
static async sendToUser(userId: string, payload: any) {
    const notification = await Notification.create({ ... });
    const targets = this.clients.filter(c => c.id === userId);

    if (targets.length > 0) {
        targets.forEach(c => c.res.write(`data:
            ${JSON.stringify(notification)}\n\n`));
    } else {
        redis.publish('NOTIFICATION_CHANNEL',
            JSON.stringify({ userId, notification }));
    }
}

// Controller heartbeat
const keepAlive = setInterval(() => res.write(':
   keep-alive\n\n'), 30000);
req.on('close', () => { clearInterval(keepAlive); ... });
```

This approach allow the system to scale horizontally while maintaining consistent push-event delivery.

### 4.3.6 Stateless Authentication via JWT

To ensure horizontal scalability of the backend, we adopted an authentication architecture based on JSON Web Tokens (JWT). In a distributed system, relying on memory-based server-side sessions would necessitate maintaining shared state across all backend instances or implementing sticky sessions at the load balancer level, limiting routing flexibility.

Each authenticated request includes a signed JWT containing user metadata (ID, permissions, admin flag). By sharing a secret key across all instances, every backend replica can independently validate tokens without querying a database or a centralized cache.

The validation process is implemented as an Express middleware that intercepts incoming requests:

```
export const authenticateToken = (req: Request, res:
   Response, next: NextFunction) => {
const authHeader = req.headers['authorization'];
const token = authHeader && authHeader.split(' ')[1];

if (!token) return res.sendStatus(401);

jwt.verify(token, process.env.JWT_SECRET as string, (err,
   user) => {
    if (err) return res.sendStatus(403);
    req.user = user as UserPayload;
    next();
});

};
```

In the event of an expired or invalid token, the system returns an appropriate HTTP status code (401 or 403), allowing the client to handle automatic logout or redirection to the Login view. This stateless approach not only reduces validation latency but also allows Traefik to load-balance traffic across replicas without dependency on session context, thus maximizing the reliability of the distributed system.

# 5 Validation

## 5.1 Automatic Testing

The project employs a three-tier testing strategy: unit tests, integration tests, and infrastructure tests. All automated tests use Vitest as the test runner and can be executed via a single command:

```
./test.sh
```

This script starts a temporary Redis container, runs all server unit and integration tests, then runs client-side unit tests, and finally cleans up the container. This ensures a reproducible, zero-configuration test environment.

### 5.1.1 Unit Tests (Server)

Unit tests validate individual components in isolation using mocked dependencies. They run without any external services (no Redis, no MongoDB).

- **Canvas Pixel Write and Boundary Validation** (`canvas.test.ts`): verifies that the `CanvasService.draw` method correctly writes valid pixels, rejects out-of-bounds coordinates (negative, exceeding width/height), and rejects invalid colour indices outside the palette range. Also tests that `markLobbyDirty` is called only for successful writes. *Tests: FR3, NFR1.*
- **Canvas Batch Processing and Conflict Resolution** (`canvas.test.ts`): tests the `drawBatch` method with mixed valid and invalid pixels, verifying that invalid entries are silently filtered and only successful updates are returned. Also tests concurrent `draw` calls to validate the last-writer-wins semantic. *Tests: FR3, NFR4.*
- **CoordinationService Distributed Lock** (`coordination.service.test.ts`): tests three scenarios: (1) lock not acquired (another instance holds it) — no flush occurs; (2) lock acquired — dirty lobbies are flushed and the lock is released; (3) database save failure — the lobby is re-added to the dirty set for retry. *Tests: NFR2, NFR4.*
- **Redis Adapter Lifecycle** (`redisAdapter.test.ts`): verifies that the pub/sub client pair is created, reused on subsequent calls, and properly cleaned up on shutdown. *Tests: NFR4.*
- **Lobby Service and Creation** (`lobby.service.test.ts`, `lobby.creation.-test.ts`): tests lobby CRUD operations and the factory method for creating lobbies with associated canvases. *Tests: FR2.*
- **Canvas Store** (`canvas.store.test.ts`): verifies the Redis-backed store operations—lobby presence checks, metadata parsing, atomic single-pixel writes via `SETRANGE`, out-of-bounds rejection, and dirty-set tracking. *Tests: FR3, NFR4.*
- **Socket Utilities** (`socket.utils.test.ts`): tests the helper that disconnects all sockets belonging to a given user within a lobby, used by the kick and duplicate-session flows. *Tests: FR4.*

### 5.1.2 Unit Tests (Client)

- **Pixel Buffer Composable** (`usePixelBuffer.test.ts`): tests the off-screen rendering pipeline using mocked Canvas API. Verifies correct ABGR Uint32 colour conversion (e.g. `#ff0000` → `0xFF0000FF`), correct 2D grid mapping, single-pixel updates, palette hot-swapping, `onBufferUpdate` callback invocation, and confirms that the optimised `putImageData` path is used instead of per-pixel `fillRect`. *Tests: NFR1.*

### 5.1.3 Integration Tests (Server)

Integration tests verify the interaction between real components connected to a live Redis instance and an in-memory MongoDB (`mongodb-memory-server`). A shared utility module (`socket-test-utils.ts`) bootstraps a real HTTP server with Socket.IO, seeds test data (users, lobbies, canvases) in both MongoDB and Redis, and provides helper functions for creating authenticated client connections. Unless otherwise noted, JWT verification is mocked so that tests exercise application behaviour rather than the authentication protocol itself.

- **Socket Authentication** (`socket-auth.test.ts`): tests that connections are rejected without a token, with an invalid token, and with an expired token, while a valid token succeeds. This is the only integration test that uses real JWT verification (no mock). *Tests: NFR3.*

- **Socket Broadcasting** (`socket-broadcasting.test.ts`): two clients join the same lobby; Client A emits `DRAW` and `DRAW_BATCH`, and Client B receives `PIXEL_UPDATE` and `PIXEL_UPDATE_BATCH` respectively with correct data. *Tests: FR3, NFR1.*

- **State Hydration** (`state-hydration.test.ts`): Client A joins a lobby, draws a pixel, and confirms the update. Client B then joins and receives an `INIT_STATE` whose pixel buffer contains Client A's pixel. This validates the full Redis-backed state persistence and hydration pipeline. Also verifies that `USER_LEFT` is correctly broadcast when Client A disconnects. *Tests: FR3, NFR2.*

- **Kick User Flow** (`socket-kick.test.ts`): the lobby owner calls `POST /api/lobbies/:id/kick`; the target user receives `FORCE_DISCONNECT` with reason `kicked`, the owner receives `USER_LEFT`, and the kicked user can successfully reconnect afterward. *Tests: FR4.*

- **Ban User Flow and Persistence** (`socket-ban.test.ts`): the owner bans a user via REST; the target receives `FORCE_DISCONNECT` with reason `banned`, the ban is persisted in MongoDB, and a subsequent reconnection attempt is rejected with the `banned` error message. *Tests: FR4, NFR3.*

- **Duplicate Session Handling** (`socket-duplicate-session.test.ts`): the same user connects twice to the same lobby; the older connection receives `FORCE_DISCONNECT` with reason `duplicate_session`. *Tests: NFR2.*
- **Admin Override Authority** (`socket-admin-override.test.ts`): a system administrator clears the canvas of a lobby they do not own (the owner receives `CANVAS_CLEARED`), and then kicks the owner via REST. This verifies that admin privileges correctly bypass ownership checks. *Tests: NFR3.*

### 5.1.4 Infrastructure Tests

Infrastructure tests verify the distributed deployment at the Docker Compose level using the production configuration. The script `run-infrastructure-tests.sh` performs:

1. Starts the full stack with 3 backend replicas behind Traefik.
2. Sends 20 HTTP requests to `/api/health` and verifies that responses come from 3 unique `serverId` values (confirming load distribution).
3. Scales up to 5 replicas and verifies distribution across all 5.
4. Scales down to 2 replicas and verifies distribution across 2.

These tests validate that Traefik correctly discovers new backend instances via the Docker API, distributes traffic across all available instances, and stops routing to removed instances—all without downtime or manual reconfiguration. *Tests: NFR4.*

## 5.2 Acceptance Test

In addition to automated tests, the SPA was deployed on a remote server with a public domain name and tested manually on different devices (desktop and mobile) under different network conditions (Wi-Fi and mobile data). Manual testing was necessary for aspects that are difficult to automate, such as visual rendering correctness, touch input behaviour, and multi-device synchronisation under realistic conditions.

- **Visual rendering correctness**: the canvas rendering pipeline (zoom, pan, grid overlay, colour accuracy) was verified by visual inspection at multiple zoom levels and canvas sizes.
- **Touch interaction on mobile**: single-finger draw vs. two-finger pinch-zoom was tested on Android and iOS devices.
- **Multi-instance synchronisation**: two users drawing simultaneously across different server instances (3 replicas behind Traefik), confirming pixel updates propagated correctly.

- **Cross-instance notifications**: a user was banned from one instance while their SSE connection was on a different instance; the notification arrived correctly via the Redis pub/sub bridge.
- **Graceful shutdown under load**: a server instance was stopped (`docker stop`) while users were drawing; canvas state was fully persisted to MongoDB.

# 6 Deployment

## 6.1 Prerequisites

The following software is required on the host machine:

- **Docker** (version 20.10 or later) with Docker Compose V2.
- **Git** for cloning the repository.

No other runtime dependencies (Node.js, MongoDB, Redis) need to be installed on the host; all services run inside Docker containers.

## 6.2 Development Environment

1. Clone the repository:

```
git clone https://github.com/pixie-git/pixie
cd pixie
```

2. Start all services:

```
docker compose up
```

3. Access the application:
   - Frontend: `http://localhost:5173`
   - API: `http://localhost:3000/api`
   - Swagger UI: `http://localhost:3000/api-docs`
   - Mongo Express: `http://localhost:8081`
   - Traefik Dashboard: `http://localhost:8080`

The development configuration uses Vite's dev server with hot module replacement for the frontend and `tsx watch` for the backend, both with source directory volume mounts for live code reloading.

| Variable | Required | Default | Description |
|---|---|---|---|
| JWT_SECRET | Yes | — | Secret key for JWT signing |
| CLIENT_ORIGIN | No | * | Allowed CORS origins |
| CLIENT_PORT | No | 3080 | Host port for entry point |
| REDIS_URL | No | redis://redis:6379 | Redis connection URL |
| CANVAS_WIDTH | No | 64 | Default canvas width |
| CANVAS_HEIGHT | No | 64 | Default canvas height |
| VITE_API_URL | No | (empty) | API URL (build-time) |

Table 1: Production environment variables.

## 6.3 Production Environment

1. Build and start the production stack:

```
JWT_SECRET=your-secret-here \
  docker compose -f docker-compose.prod.yml up -d --build
```

2. Access the application at `http://localhost:3080`. All routes (frontend, API, WebSocket, Swagger) are served through a single Traefik entry point.

3. To scale the backend horizontally:

```
docker compose -f docker-compose.prod.yml up -d --scale
   server=3
```

## 6.4 Environment Variables

## 6.5 Docker Multi-Stage Builds

Both the server and client use multi-stage Dockerfiles with three stages:

- **dev-stage**: installs all dependencies (including devDependencies) and runs the development server with live reloading. Used by the development Compose file via `target: dev-stage`.
- **build**: extends the dev stage and runs the TypeScript/Vite compilation to produce optimised production assets.
- **production-stage**: for the server, copies only the compiled JavaScript and production dependencies onto a clean Node.js Alpine image. For the client, copies the compiled static assets onto an Nginx Alpine image. This minimises the final image size and attack surface.

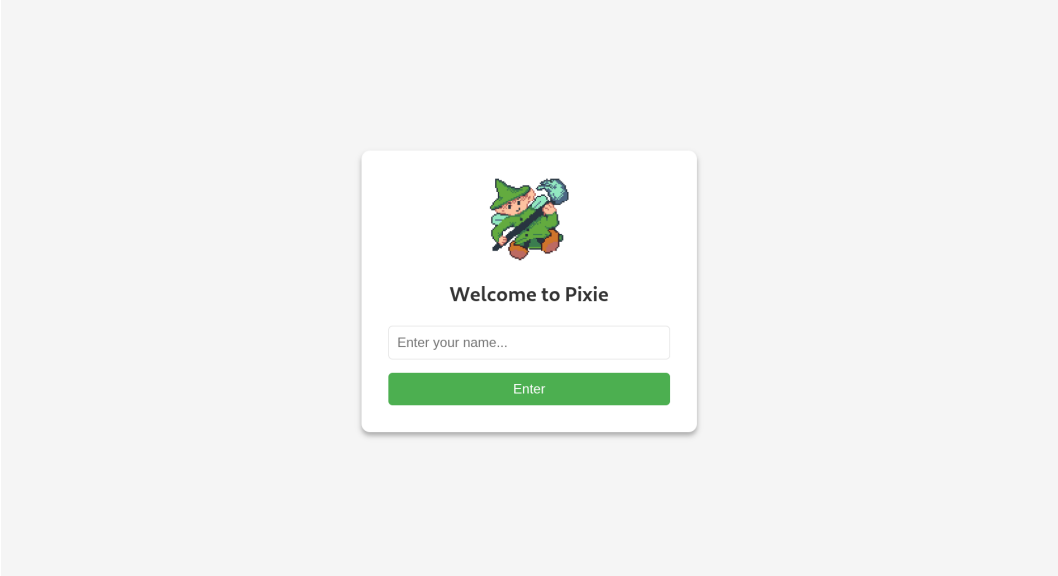


Figure 5: Login screen.

# 7 User Guide

## 7.1 Login

Upon visiting the application, the user is presented with a login screen. Enter a username and click "Login". If the username does not exist, a new account is created automatically. No password is required.

## 7.2 Browsing and Joining Lobbies

After login, the user sees the lobby list. Each lobby shows its name and owner. Click on a lobby to join and enter the collaborative drawing session.

## 7.3 Creating a Lobby

Click the "Create" button to open the lobby creation form. Configure the lobby name, optional description, canvas dimensions (width and height in pixels), maximum number of collaborators, and the colour palette. Click "Create" to launch the new lobby.

## 7.4 Drawing on the Canvas

The pixel art editor is the core of the application:

- **Drawing**: select a colour from the palette bar, then click or drag on the canvas to paint pixels. Strokes are interpolated using Bresenham's algorithm for smooth lines.

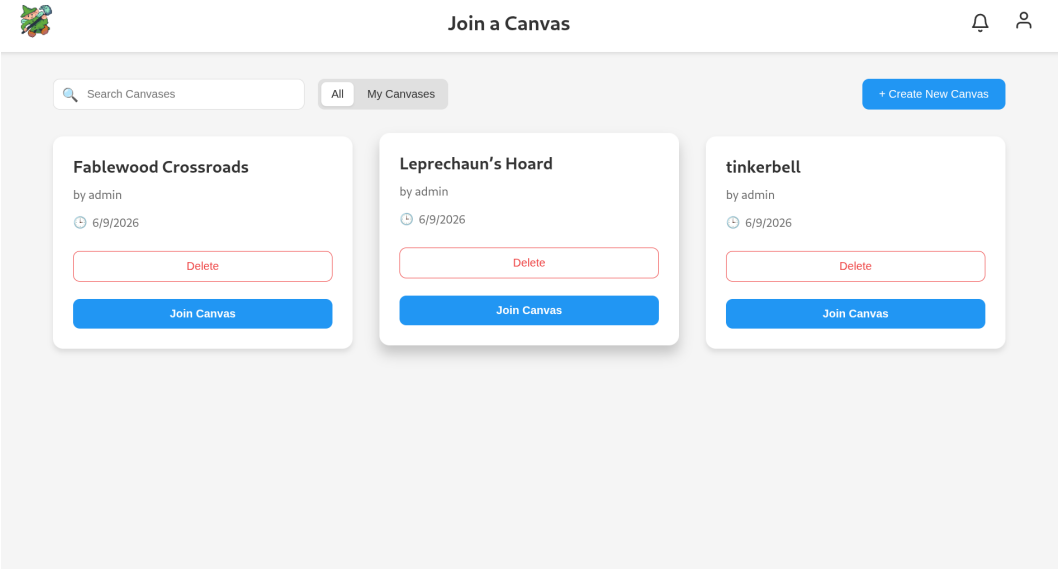


Figure 6: Lobby browser with available rooms.

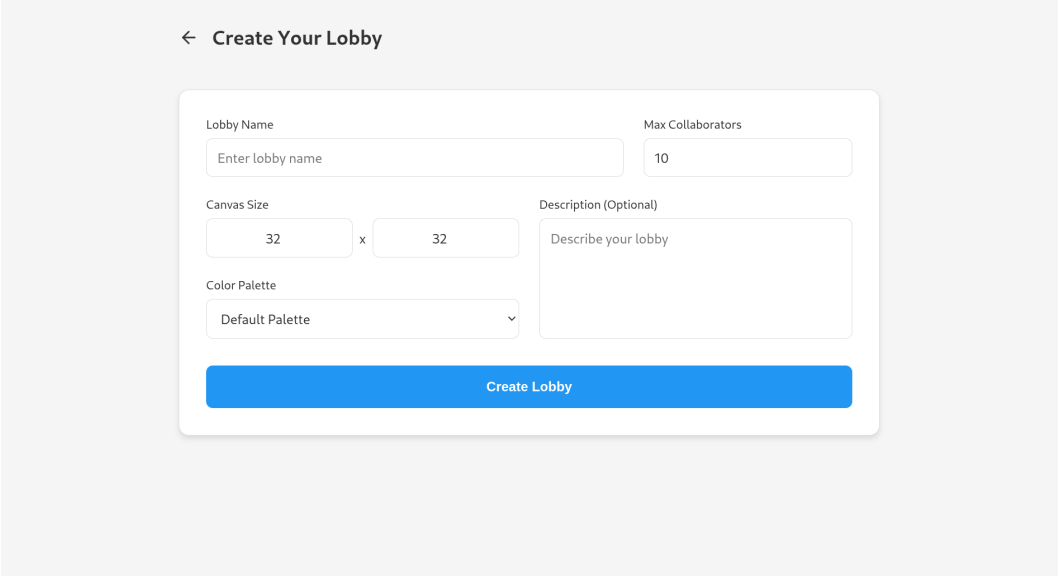


Figure 7: Lobby creation form with canvas configuration.


Figure 8: Pixel art editor with palette, zoom controls, and user list.

- **Zooming**: scroll the mouse wheel to zoom in/out. On touch devices, use the pinch gesture.
- **Panning**: hold Alt + click and drag (mouse), or use two-finger drag (touch) to navigate the canvas.
- **Grid**: a pixel grid overlay appears automatically when the zoom level exceeds 4×.
- **Connected users**: the sidebar displays a real-time list of users currently in the lobby.
- **Export**: click the export button to download the current canvas as a PNG image.

## 7.5 Moderation (Lobby Owner)

Lobby owners can manage participants through the user list sidebar:

- **Kick**: removes a user from the lobby. The user can rejoin.
- **Ban**: removes a user and prevents them from rejoining. The banned user receives a notification. Owners can manage the ban list and unban users.
- **Clear canvas**: resets all pixels to the default colour (index 0). This action is broadcast to all connected users.

System administrators have the same moderation capabilities across all lobbies.

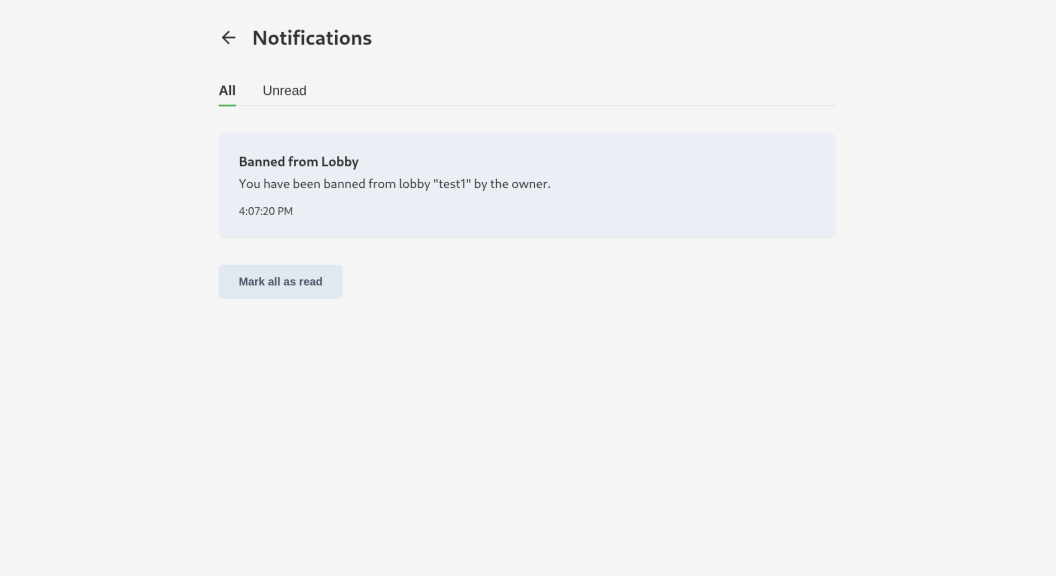


Figure 9: Notification panel with unread indicators.

### 7.6 Notifications

Notifications are delivered in real time via the bell icon in the navigation bar. The unread count badge updates automatically. Clicking the bell opens the notification panel where users can view and dismiss notifications.

### 7.7 User Profile

Click on the username in the navigation bar to access the profile page. Users can change their username from this screen. The change is reflected immediately across the application.

## 8 Self-evaluation

### 8.1 Andrea Zavatta

**Role.** My primary responsibility within the group was frontend development and the implementation of backend features. I built the login interface and the first pixel canvas prototype. Later, I implemented the notification system end-to-end: the SSE streaming endpoint, the MongoDB persistence layer, and eventually the Redis pub/sub bridge that allows notifications to be delivered to users whose SSE connection lands on a different backend replica. I also designed and implemented the moderation UI (kick, ban, clear canvas), the lobby browser, the user profile and username-change flow, and the centralized error-handling mechanism built around a Pinia notification store and a global popup component. On the backend I contributed the JWT authentication

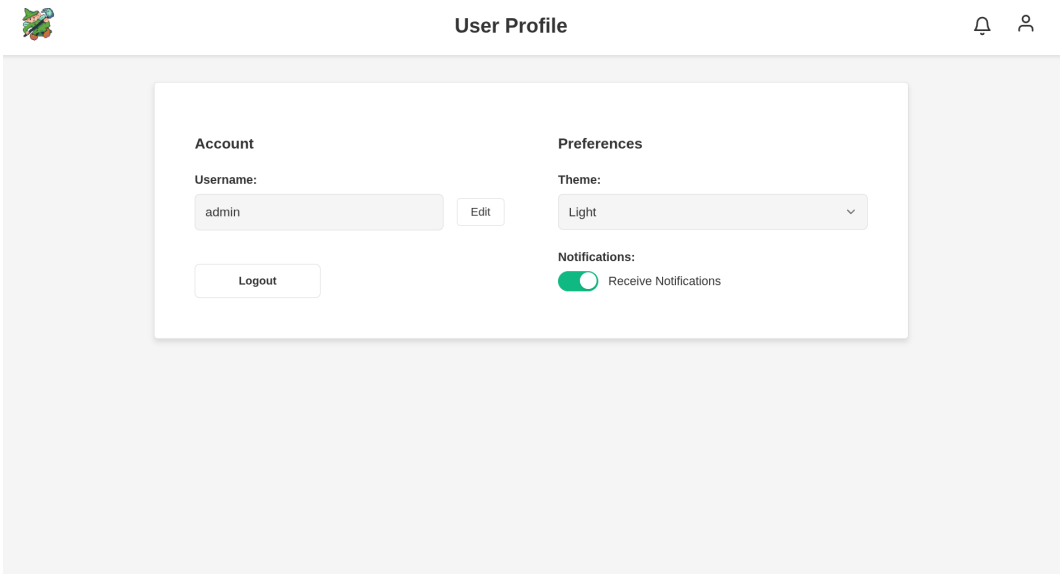


Figure 10: User profile page.

middleware, the permission middleware for role-based access control, and parts of the OpenAPI documentation. In the testing phase I wrote the integration tests covering socket authentication, the kick and ban flows, and the admin override authority.

**Strengths of the product.**

- **Horizontal Scalability:** By leveraging Redis for shared collaborator counts, Socket.io clustering via the Redis Adapter, and Redis Pub/Sub channels for SSE notifications, the application can scale across multiple server nodes behind a load balancer without causing user state desynchronization.
- **Unified Error Management:** The combination of backend `AppError` normalization and client-side Axios interception ensures the application degrades gracefully under failure, presenting uniform and safe error descriptions to users regardless of which server instance processes the request.
- **Stateless and Secure Session Lifecycle:** Utilizing JWTs at both the HTTP and WebSocket layers eliminates session synchronization requirements across servers, enabling a lightweight authentication architecture that does not depend on sticky sessions or shared session stores.

**Weaknesses of the product.**

- **JWT Storage Vulnerability**: The current implementation stores authentication tokens in localStorage, rendering them susceptible to Cross-Site Scripting (XSS) attacks. Because localStorage is accessible to any JavaScript executing in the

browser context, a malicious script could extract valid user sessions. While a 7-day expiry provides a convenient user experience, it amplifies the window of opportunity for an attacker if a token is compromised. Transitioning to HttpOnly, Secure, and SameSite=Strict cookies would mitigate this by preventing client-side JavaScript from accessing the session data, though this would necessitate a robust CSRF protection strategy.

- **Absence of Rate Limiting:** The API currently lacks request rate-limiting mechanisms, leaving authentication endpoints and canvas drawing channels susceptible to abuse, automated scripting, or denial-of-service attempts. Implementing a token-bucket or sliding-window rate limiter (e.g., via `express-rate-limit` backed by Redis) would mitigate this risk across all server instances.

## 8.2 Matteo Susca

**Role.** My responsibilities spanned both the frontend canvas subsystem and the backend distributed infrastructure. On the frontend, I owned the canvas layer: the dual-canvas rendering architecture (off-screen pixel buffer decoupled from the visible display canvas), the pixel rendering pipeline based on a `Uint32Array` lookup table for $O(n)$ full-canvas fills, and all navigation logic including mouse wheel zoom, Alt-drag pan, and touch gesture handling (single-finger draw vs. two-finger pinch-zoom and pan). On the backend, I designed and implemented the Redis-backed `CanvasStore` (binary pixel storage, dirty tracking, atomic byte-offset writes), the `CoordinationService` (distributed flush lock with Lua-based safe release, write-behind persistence), and the Redis adapter integration for cross-instance Socket.IO broadcasting. I was also responsible for the production deployment stack (Traefik as load balancer and service discovery layer, multi-stage Docker builds, dynamic replica scaling), the integration test framework (`socket-test-utils`, `mongodb-memory-server` integration, the automated `test.sh` script), and the infrastructure tests that verify Traefik's dynamic backend discovery under scaling events.

**Strengths.** The component I am most satisfied with is the canvas rendering pipeline. The two-canvas architecture cleanly separates pixel data management from visual display: the off-screen buffer holds the pixel-accurate state at 1:1 scale and is updated through direct Uint32Array writes, while the visible canvas is redrawn with a single `drawImage` call that applies the current zoom and pan transform. This means that the most frequent user interactions—zooming and panning—never touch pixel data at all, keeping them consistently fast regardless of canvas size. The second area I am satisfied with is the production deployment stack. Traefik's integration with the Docker socket API for automatic backend discovery means that scaling the server up or down requires a single `--scale` flag, with no manual configuration reload or downtime; the infrastructure tests validate this behavior automatically. Finally, the touch input system required careful design to disambiguate single-finger drawing from two-finger pinch-zoom and pan, which are conflicting gesture interpretations. Implementing this through dedicated composables (`useCanvasTransform`, `useInputHandler`) with explicit gesture state tracking produced

a reliable and responsive experience on both Android and iOS without interfering with the desktop mouse flow.

**Weaknesses.** The most significant architectural weakness is that Redis remains a single point of failure. The distributed lock, canvas state, pub/sub channels, and lobby capacity counters all depend on a single Redis instance; if it becomes unavailable, drawing operations and lobby joins fail immediately. The correct long-term solution (Redis Sentinel or Redis Cluster) was deferred and discussed in the future works section, but the current deployment is not production-resilient in this regard. A second significant limitation is the lack of scalability for large canvas dimensions. The current design allocates a contiguous binary buffer of `width` × `height` bytes in Redis and transmits the full buffer to every client on join. For a $2048 \times 2048$ canvas this amounts to over 4 MB transferred on every `JOIN_LOBBY` event and held entirely in Redis memory per active lobby, which is not viable in practice. Addressing this would require replacing the flat buffer with a chunked or sparse representation, streaming only the visible viewport region to the client, and adapting the rendering pipeline to composite chunks on demand—a substantial architectural change that was out of scope for this project.

# 9 Future Works

- **Undo/Redo**: a per-user action history allowing users to undo recent strokes. This would require a command log per session with inverse operations. The main challenge is handling concurrent undos when multiple users have modified the same pixels.
- **Drawing tools**: extend the editor with geometric primitives (lines, rectangles, circles, fill bucket). These would be computed client-side and emitted as batch pixel updates, requiring no protocol changes.
- **Unlimited / dynamically-resizable canvas**: allow the canvas to grow beyond its initial dimensions as users draw near the edges. This would require replacing the fixed-size binary buffer with a sparse representation (e.g. a hash of chunk coordinates to chunk buffers) and adapting the rendering pipeline to composite visible chunks on demand.
- **Timelapse and replay**: persist a log of all drawing events (with timestamps) to enable playback of the canvas creation process. This could be implemented as an append-only log in MongoDB, replayed client-side with configurable speed.
- **Private lobbies with invitations**: add password-protected or invite-only lobbies. The ban list infrastructure already supports access control; this would extend it with an allow-list and optional password verification on join.
- **Redis Cluster / Sentinel**: the current deployment uses a single Redis instance, which is a single point of failure. Integrating Redis Sentinel for automatic failover

or Redis Cluster for data sharding would improve production resilience. The application code would require minimal changes as the Redis client library supports cluster mode natively.