CollabBoard Pre-Search Document

Phase 1: Define Your Constraints

1. Scale & Load Profile

Decision: Design for 5-20 concurrent users per board, 1-5 boards active simultaneously, ~50 total registered users at launch. In 6 months: irrelevant — this is a portfolio/program project, not a startup.

Why: This is a Gauntlet program deliverable with a 7-day build window. The performance targets (5+ concurrent users, 500+ objects, 60 FPS) define the real ceiling. Designing beyond that is wasted time.

Traffic pattern: Spiky. Demo-driven — load hits when you present or when evaluators test it. Zero traffic otherwise.

Real-time requirements: Yes, WebSockets are non-negotiable. The spec demands <100ms object sync and <50ms cursor sync. SSE won't cut it for bidirectional cursor streaming. Use a single WebSocket connection per client multiplexing all message types as frames.

Cold start tolerance: Moderate. A 2-3 second cold start on first board load is acceptable. Evaluators won't be hitting it cold repeatedly.

Key tradeoff: Ignoring horizontal scaling entirely. Single-process WebSocket state means one server handles all connections. This breaks at ~100 concurrent users. That's fine — we'll never hit it.

2. Budget & Cost Ceiling

Decision: $100/month budget ceiling. Spend where it buys time.

Why: $100/mo unlocks real infrastructure without overthinking free-tier limits. This removes friction from hosting, database, and LLM usage decisions.

Where to spend:

- **LLM calls ($20-40/mo):** Use Claude Sonnet freely during dev and demo without worrying about token counts. Budget for heavy iteration on AI agent prompts.
- **Deployment ($5-10/mo):** Fly.io dedicated machine. No cold starts, no free-tier resource limits, reliable WebSocket connections.
- **Database ($0-25/mo):** Neon Pro or Supabase Pro if free tier hits limits. Otherwise stay on free tier and pocket the savings.
- **Buffer (~$30-50/mo):** Headroom for unexpected API usage spikes during demo/evaluation.

Key tradeoff: Spending money on infrastructure means fewer constraints to work around. The $100/mo ceiling is generous for a demo project — most of it will go to LLM API calls during development.

3. Time to Ship

Decision: 24-hour MVP, then 6 remaining days for full feature set. Speed-to-market is the only priority. Maintainability is irrelevant.

Why: The 24-hour MVP gate is a hard requirement. Ship or fail. After that, ~6 days to layer on connectors, frames, multi-select, AI agent, and polish. There is no "long-term" — this project's useful life is the evaluation period.

Concrete time allocation (MVP, 24 hours):

Block	Hours	Deliverable
Hours 0-3	3h	Cargo project, Frame type, Postgres schema + migrations, AppState
Hours 3-5	2h	GitHub OAuth (`/api/auth/*`) + session management + `/api/ws` frame relay
Hours 5-6	1h	**Tests: frame serialization, session lifecycle, WS connect**
Hours 6-9	3h	Board + Object services (CRUD, in-memory state, Postgres persist)
Hours 9-10	1h	**Tests: object CRUD, board state hydration, LWW**
Hours 10-14	4h	React+Konva frontend: canvas, sticky notes, shapes, toolbar
Hours 14-18	4h	Frame client (WS), real-time sync, cursor broadcast, presence UI
Hours 18-19	1h	**Tests: frame client dispatch, multi-client sync smoke test**
Hours 19-22	3h	Deploy to Fly.io, integration testing, bug fixes
Hours 22-24	2h	Buffer for the inevitable fire

Post-MVP (Days 2-7):

Block	Hours	Deliverable
Days 2-3	16h	Connectors, text, multi-select, rotate, board CRUD UI
Days 4-5	16h	AI agent subsystem (LLM tool calling, 6+ commands)
Days 6-7	16h	Copy/paste, disconnect recovery, polish, performance

Testing approach: Unit tests (happy path + smoke) are woven into the MVP build at 3 checkpoints — after backend core, after object subsystem, and after real-time sync. Each test block is 1 hour. Tests cover WS dispatch routing, frame serialization, object CRUD, LWW conflict resolution, session lifecycle, and a multi-client sync smoke test. This is the minimum needed for confidence without slowing the sprint.

Key tradeoff: 3 hours of testing in a 24-hour MVP is tight but non-negotiable. Prior's proven patterns reduce the risk of architectural surprises, making this feasible.

---

4. Compliance & Regulatory Needs

Decision: None. Zero compliance work.

Why: Program project, not a production SaaS. No real user data beyond demo accounts. Every hour spent on compliance is an hour stolen from features that get evaluated.

What you actually do:

- Don't commit secrets to git.
- That's it. No local auth, no passwords to hash.

Key tradeoff: If an evaluator asks "what about GDPR?" the answer is "this is a technical demo — here's where I'd add consent management and data deletion endpoints." Knowing the answer is more valuable than implementing it.

---

5. Team & Skill Constraints

Decision: Solo build. Rust backend (porting proven patterns from Prior), React + Konva.js frontend (new, no prior experience with Konva).

Stack divergence from CLAUDE.md: The root `CLAUDE.md` specifies Bun + Hono + htmx + SQLite as the default Gauntlet stack. CollabBoard intentionally diverges because: (1) the Prior project provides battle-tested Rust/Axum patterns for frame-based WebSocket protocols — porting is faster than building from scratch; (2) a real-time collaborative canvas genuinely needs rich client-side interactivity (React + Konva), not server-rendered HTML fragments; (3) PostgreSQL handles concurrent WebSocket-driven writes better than SQLite. The tradeoff is slower iteration speed in Rust vs TypeScript, offset by proven architecture.

Why: The Prior project provides a battle-tested monokernel, frame protocol, and Axum gateway pattern. Porting these to a new domain (whiteboard) is faster than building from scratch in any stack — the architecture is already debugged.

Frontend rationale (Konva.js): No prior experience with Konva, but it's the right choice for iteration speed. Konva gives the hardest whiteboard primitives for free: drag-and-drop, `Transformer` (resize/rotate handles), hit detection, text editing overlays. `react-konva` means writing React components that render to canvas — not a new paradigm. The API surface is small (~10 components). The alternative (raw Canvas 2D) would add 2+ days of boilerplate for features Konva provides out of the box.

Key tradeoff: Rust is slower to iterate on than TypeScript for the backend, but the patterns are already proven and the type system catches bugs at compile time. Konva is new but the learning curve is small relative to the time it saves on whiteboard UI primitives.

Phase 2: Architecture Discovery

6. Hosting & Deployment

Options considered:

1. **Fly.io** — Docker containers, native WebSocket support, long-lived processes, built-in scaling.
2. **Railway** — Simple container hosting, easy deploys, but less control.
3. **VPS** — Full control, cheap, but manual ops burden.

Decision: Fly.io, `dfw` region (Dallas — evaluators are Austin-based).

Why: Purpose-built for long-lived WebSocket connections on container infrastructure with zero DevOps overhead. `fly deploy` from a Dockerfile gives you TLS and health checks in one command. Rust compiles to a static binary — the final Docker image is ~10-20MB. Dedicated machine ($5-10/mo) eliminates cold starts and free-tier resource limits.

Static assets: Axum serves the built React SPA from an embedded directory using `tower-http::services::ServeDir`. Gzip compression via `tower-http::compression::CompressionLayer`. Cache headers: `Cache-Control: public, max-age=31536000, immutable` for hashed assets, `no-cache` for `index.html`. Single binary deployment — no separate CDN needed.

Why not DigitalOcean: App Platform supports Docker but has worse WebSocket support (idle timeouts, no persistent connections). Fly is purpose-built for this.

CI/CD: None for a 7-day sprint. `fly deploy` from the terminal.

Key tradeoff: Locked into Fly's orchestration. Acceptable for a demo.

---

7. Authentication & Authorization

Decision: GitHub OAuth from day one. Google OAuth as secondary if time allows. No local auth (no passwords). Token-based sessions.

Why: The MVP requires auth. OAuth eliminates password hashing, email verification, and forgot-password flows — all time sinks. GitHub OAuth is one redirect flow: `GET /api/auth/github` -> GitHub authorize -> callback -> create/find user -> create session token -> return token to client -> redirect to app. This is ~2 hours of work including the user table and session management.

Session mechanism: Opaque session tokens stored server-side in Postgres.

Context	Mechanism
REST API	`Authorization: Bearer <token>` header
WebSocket upgrade	Short-lived upgrade ticket: client calls `POST /api/auth/ws-ticket` (authenticated), receives a single-use ticket valid 30 seconds. WS connects via `GET /api/ws?ticket=<ticket>`. Server validates + deletes ticket on upgrade.
Logout	`POST /api/auth/logout` with Bearer token — deletes session from Postgres

Why not cookies: Token-based is more explicit for a SPA. No CSRF concerns. The WS upgrade ticket pattern avoids exposing the long-lived session token in a URL query parameter.

Implementation:

- `GET /api/auth/github` — redirect to GitHub OAuth authorize URL

- `GET /api/auth/github/callback` — exchange code for token, upsert user, create session, return token to client
- `GET /api/auth/me` — return current user from session (Bearer token)
- `POST /api/auth/logout` — delete session
- `POST /api/auth/ws-ticket` — generate single-use upgrade ticket (30s TTL)

Access model: Public-by-link. Anyone with the board UUID can join and collaborate. No `board_members` table, no invite tokens. Board creator is the owner (can delete the board). This is the simplest model that works for a demo — identical to sharing a Google Doc link.

Key tradeoff: GitHub OAuth requires a GitHub OAuth App registration (takes 2 minutes). Users without GitHub accounts can't use the app at MVP — acceptable for a developer-audience demo. Google OAuth added post-MVP if time allows.

---

8. Database & Data Layer

Options considered:

1. **SQLite (like Prior)** — Zero infrastructure, but no concurrent write support from multiple connections.
2. **PostgreSQL (managed, Neon/Supabase)** — Proper concurrent writes, JSONB for flexible object storage, full SQL.
3. **Redis + PostgreSQL** — Redis for ephemeral state, Postgres for durable.

Decision: PostgreSQL (Neon, `us-east` region) + in-memory board state.

Why: Postgres gives proper concurrent writes, JSONB columns for flexible board object properties, and a migration path to production. The `frames` table (append-only frame log, ported from Prior) provides audit trail and history replay. Live board state still lives in-memory for performance — Postgres is the durable backing store, not the hot path.

Why Neon over Supabase: Neon is pure Postgres with zero extra services — lighter, faster to provision, has database branching for dev workflows. Supabase bundles auth/storage/REST that we won't use. ~20-30ms round-trip from Dallas to `us-east` is fine since the DB is off the hot path (in-memory state handles real-time reads).

Schema Design

```
-- Append-only frame log (ported from Prior's frame_db)
CREATE TABLE frames (
    seq         BIGSERIAL PRIMARY KEY,
    ts          BIGINT NOT NULL DEFAULT (EXTRACT(EPOCH FROM now()) * 1000),
    id          UUID NOT NULL,
    parent_id   UUID,
    syscall     TEXT NOT NULL,
    status      TEXT NOT NULL,
    board_id    UUID,
    "from"      TEXT,
    data        JSONB NOT NULL DEFAULT '{}'
);
CREATE INDEX idx_frames_board_seq ON frames(board_id, seq);

-- Sessions (server-side, token-based)
CREATE TABLE sessions (
    token       TEXT PRIMARY KEY,            -- opaque random token (32 bytes, hex-encoded)
    user_id     UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    expires_at  TIMESTAMPTZ NOT NULL DEFAULT (now() + INTERVAL '30 days')
);
CREATE INDEX idx_sessions_user ON sessions(user_id);
CREATE INDEX idx_sessions_expires ON sessions(expires_at);
```

```
-- WS upgrade tickets (single-use, short-lived)
CREATE TABLE ws_tickets (
    ticket      TEXT PRIMARY KEY,            -- opaque random token
    user_id     UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    expires_at  TIMESTAMPTZ NOT NULL DEFAULT (now() + INTERVAL '30 seconds')
);

-- Relational tables
CREATE TABLE users (
    id              UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    github_id       BIGINT UNIQUE NOT NULL,  -- GitHub user ID for OAuth upsert
    name            TEXT NOT NULL,
    avatar_url      TEXT,                      -- GitHub avatar
    color           TEXT NOT NULL DEFAULT '#4CAF50',
    created_at      TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE boards (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name        TEXT NOT NULL,
    owner_id    UUID NOT NULL REFERENCES users(id),
    created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE board_objects (
    id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    board_id    UUID NOT NULL REFERENCES boards(id) ON DELETE CASCADE,
    kind        TEXT NOT NULL,  -- sticky_note, rectangle, ellipse, line, connector, text, frame
    x           DOUBLE PRECISION NOT NULL DEFAULT 0,
    y           DOUBLE PRECISION NOT NULL DEFAULT 0,
    width       DOUBLE PRECISION,
    height      DOUBLE PRECISION,
    rotation    DOUBLE PRECISION NOT NULL DEFAULT 0,
    z_index     INTEGER NOT NULL DEFAULT 0,
    props       JSONB NOT NULL DEFAULT '{}',  -- color, text, points, etc.
    created_by  UUID REFERENCES users(id),
    updated_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    version     INTEGER NOT NULL DEFAULT 1    -- monotonic, for LWW ordering
);
CREATE INDEX idx_board_objects_board ON board_objects(board_id);
```

Frames table notes: Append-only audit log. No retention policy for MVP — expected max ~100K frames over the project's lifetime. The `idx_frames_board_seq` index supports the primary query pattern: `SELECT * FROM frames WHERE board_id = ? AND seq > ? ORDER BY seq` (for replay/recovery post-MVP).

Real-Time Sync Approach

Last-write-wins (LWW) with object-level granularity. Not CRDTs, not OT.

- **Server is the clock.** Server timestamps every mutation. Client `ts` field is ignored on inbound frames — the server assigns `ts` from its own monotonic clock. This eliminates clock skew issues entirely.
- **Per-object versioning.** Each `board_object` has a `version` column (monotonic integer, incremented on every update). When the server receives an `object:update`, it compares the incoming `version` against the current version. If incoming `version < current version`, the update is stale and dropped. This handles out-of-order delivery from the debounced persistence layer.
- If two users drag the same object simultaneously, the last mutation to reach the server wins. This is how Miro works.
- Cursor positions and presence are purely ephemeral — broadcast and forget, never persisted.

Persistence Strategy

Data	Storage	Lifetime
Cursor positions	In-memory, broadcast via WS frames	Ephemeral
Presence (who's online)	In-memory HashMap per board	Ephemeral
Board objects (live)	In-memory HashMap per board	Duration of active session
Board objects (durable)	Postgres `board_objects` table	Permanent
Frame audit log	Postgres `frames` table	Permanent
Users, boards metadata	Postgres relational tables	Permanent
Sessions	Postgres `sessions` table	30-day TTL
WS upgrade tickets	Postgres `ws_tickets` table	30-second TTL

Debounced persistence (1s buffer):

On mutation: update in-memory state immediately, mark object as dirty, broadcast frame to peers. A background task flushes dirty objects to Postgres every 1 second (batch `UPDATE` or `INSERT ... ON CONFLICT`). On last user disconnect from a board: final synchronous flush of all dirty objects.

Crash behavior: On server crash, up to 1 second of mutations may be lost. The in-memory state is gone, but the last flushed state in Postgres is intact. On restart, boards are rehydrated from `board_objects`. This is acceptable for a 5-20 user demo — the window of data loss is small and the scenario (server crash) is rare.

Key tradeoff: Debounced persistence trades up to 1s of data loss on crash for significantly reduced Postgres write load. For a demo with 5-20 users, this is the right call.

---

9. Backend / API Architecture

Decision: Traditional Axum route handlers with shared services. Frames on the WebSocket wire, but internally it's standard async fn handlers and shared state. No kernel/subsystem abstraction — that's an unnecessary layer for this scale.

Why: For a 24-hour MVP, the kernel scaffold (channel routing, subsystem registration, dispatch loop) adds ~4 hours of infrastructure before any business logic. Standard Axum with shared services takes ~1 hour to scaffold and is immediately readable. The frame protocol on the wire gives the same client-side architecture regardless — the kernel is an internal concern.

Frame Protocol (wire format)

```
pub struct Frame {
    pub id: Uuid,
    pub parent_id: Option&lt;Uuid&gt;,
    pub ts: i64,                          // millis since epoch (server-assigned)
    pub syscall: String,                  // "object:create", "cursor:move", etc.
    pub status: Status,                   // request | item | done | error | cancel
    pub board_id: Option&lt;Uuid&gt;,
    pub from: Option&lt;String&gt;,                   // user attribution
    pub data: HashMap&lt;String, serde_json::Value&gt;,
}
```

Status lifecycle (identical to Prior):

- `Request -> Item* -> Done` (success with 0+ results)
- `Request -> Error` (failure)
- `Cancel` (abort by parent_id)

Server Architecture

```
Axum Router
■■■ /api/auth/github           GET  -&gt; redirect to GitHub OAuth
■■■ /api/auth/github/callback  GET  -&gt; exchange code, upsert user, set session
■■■ /api/auth/me               GET  -&gt; current user from session (Bearer token)
■■■ /api/auth/logout           POST -&gt; delete session
■■■ /api/auth/ws-ticket        POST -&gt; generate single-use WS upgrade ticket
■■■ /api/ws                    GET  -&gt; WebSocket upgrade (ticket-based)
■
WebSocket Handler (frame relay)
■■■ on_connect  -&gt; validate ticket, resolve user, send session:connected
■■■ on_message  -&gt; parse Frame, dispatch by syscall prefix:
■   ■■■ "board:*"   -&gt; board_service.*()
■   ■■■ "object:*"  -&gt; object_service.*()
■   ■■■ "cursor:*"  -&gt; cursor broadcast (in-memory only)
■   ■■■ "ai:*"      -&gt; ai_service.prompt()
■■■ on_close    -&gt; remove from presence, notify peers
■
Shared Services (injected via Axum state)
■■■ BoardService    { pool: PgPool, boards: Arc&lt;RwLock&lt;HashMap&lt;Uuid, BoardState&gt;&gt;&gt; }
■■■ ObjectService   { pool: PgPool, ... }
■■■ CursorService   { boards: Arc&lt;RwLock&lt;...&gt;&gt; }  // ephemeral only
■■■ AiService       { llm_client: Box&lt;dyn LlmClient&gt;, ... }
■■■ SessionService  { pool: PgPool }
```

API Surface

`/api/auth/*` — GitHub OAuth flow + session management:

- `GET /api/auth/github` — redirect to GitHub authorize URL
- `GET /api/auth/github/callback` — exchange code, upsert user, create session, return token
- `GET /api/auth/me` — return current user (requires `Authorization: Bearer <token>`)
- `POST /api/auth/logout` — delete session (requires `Authorization: Bearer <token>`)
- `POST /api/auth/ws-ticket` — generate single-use WS upgrade ticket (requires `Authorization: Bearer <token>`)

`/api/ws` — Single WebSocket endpoint, bidirectional frames:

```
Client connects: GET /api/ws?ticket=&lt;upgrade_ticket&gt;
Server sends:    { status: "item", syscall: "session:connected", data: { user_id: "uuid", name: "Ian" } }

Client sends:    { syscall: "board:join", data: { board_id: "uuid" } }
Server sends:    { status: "item", syscall: "board:state", data: { objects: [...], users: [...] } }
Server sends:    { status: "done", syscall: "board:join" }

Client sends:    { syscall: "object:create", data: { kind: "sticky_note", x: 100, y: 200, props: { text: "Hello", color:
Server broadcasts: { status: "item", syscall: "object:created", data: { id: "uuid", kind: "sticky_note", ... } }

Client sends:    { syscall: "cursor:move", data: { x: 450, y: 300 } }
Server broadcasts: { status: "item", syscall: "cursor:moved", data: { user_id: "uuid", x: 450, y: 300 } }

Client sends:    { syscall: "ai:prompt", data: { prompt: "Create a SWOT analysis" } }
Server streams:  { status: "item", syscall: "object:created", data: { ... } }  // repeated per object
Server sends:    { status: "done", syscall: "ai:prompt" }
```

Reconnect / Recovery

MVP: On disconnect, client reconnects with a new WS upgrade ticket and re-sends `board:join`. Server responds with a full `board:state` snapshot (all current objects + presence). No incremental replay — the full snapshot is small enough for 500 objects (~50-100KB JSON).

Post-MVP: Incremental recovery using `frames.seq`. Client tracks its last-seen `seq`. On reconnect, sends `board:join` with `{ board_id, since_seq: 12345 }`. Server replays frames with `WHERE board_id = ? AND seq > ? ORDER BY seq`. If gap is too large (>1000 frames or >5 minutes), fall back to full snapshot.

Ordering guarantees: Frames are ordered by server `seq` (Postgres `BIGSERIAL`). Clients process frames in order. If a frame arrives out of order (shouldn't happen over a single TCP connection, but possible after reconnect), the per-object `version` field allows the client to discard stale updates.

AI Agent Integration

The AI service is called from the WS handler when `syscall` starts with `ai:`. It:

1. Reads current board state from shared `BoardState`
2. Calls LLM with board state + user prompt + tool definitions
3. For each tool call returned, calls `object_service` directly (same shared state)
4. Each mutation updates in-memory state, marks dirty for persistence, broadcasts frame to peers
5. When all tool calls complete, sends `done` frame for the original `ai:prompt`

The AI agent's mutations flow through the same code path as human mutations — `object_service.create()`, `object_service.update()`, etc.

Syscalls

Prefix	Syscalls	Description
`session`	`connected`	Server -> client on WS connect
`board`	`join`, `part`, `create`, `list`, `get`, `delete`, `state`	Board lifecycle and CRUD
`object`	`create`, `created`, `update`, `updated`, `delete`, `deleted`, `lock`, `unlock`	Board object mutations + broadcasts
`cursor`	`move`, `moved`	Ephemeral cursor position (send vs broadcast naming)
`ai`	`prompt`	Natural language -> tool calls -> board mutations

Key tradeoff: Flat dispatch in the WS handler is faster to build but may grow unwieldy. If dispatch logic exceeds ~300 lines, extract into a dispatcher module with per-prefix handler functions.

---

10. Frontend Framework & Rendering

Decision: React (Vite SPA) + Konva.js (via react-konva)

Why React: The app shell (toolbar, sidebar, board list, presence indicators, AI chat panel) is standard UI work. React ships that fast. This project genuinely needs rich client-side interactivity (collaborative canvas with drag/drop/resize/real-time sync), which is why we diverge from the htmx default.

Why Konva: Sweet spot for a whiteboard. Scene graph, built-in drag-and-drop, `Transformer` (resize/rotate handles out of the box), hit detection, event bubbling, text editing. Performance is solid for 500 objects.

State Management

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■  Zustand Store (board state)          ■
■  - objects: Map&lt;id, BoardObject&gt;        ■
■  - presence: Map&lt;userId, CursorPos&gt;      ■
■  - selection: Set&lt;objectId&gt;             ■
■  - viewport: { x, y, scale }          ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■  Frame Client (WebSocket singleton)     ■
■  - receives server frames               ■
■  - updates Zustand store by syscall     ■
■  - sends request frames to server       ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■  React renders from Zustand             ■
■  Konva Stage reads objects from store   ■
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

The Frame Client speaks the same protocol as the server. Incoming frames are dispatched by `syscall` to store update handlers. Outgoing user actions are serialized as request frames.

Zustand over Redux/MobX: minimal boilerplate, works outside React components (the frame client can update the store directly), selectors prevent unnecessary re-renders.

60 FPS Strategy

1. **Layer separation:** Static objects on one layer, actively-dragged object on another, cursors on a third.
2. **Viewport culling:** Only render objects within the visible viewport.
3. **Throttle cursor broadcasts:** 30 Hz max (every 33ms). Render remote cursors with CSS transforms (DOM overlay), not on Konva canvas.
4. **Batch store updates:** Accumulate incoming frames for 16ms (one animation frame), then apply all at once.
5. `React.memo` **everything:** Each canvas object component memoized.

Key tradeoff: Konva's performance ceiling is lower than PixiJS (~1000-2000 objects vs. ~50,000). For 500 objects this is fine.

---

11. Third-Party Integrations

Decision: Multi-provider LLM adapter supporting Anthropic + OpenAI formats. Provider and model configured via env/config. Default: Claude Sonnet. Prior has reference implementations for both providers.

Why: Provider lock-in is unnecessary when the adapter pattern is straightforward. Prior already has working Anthropic (`llm/client.rs`) and OpenAI (`llm/openai_client.rs`) clients behind a trait interface. Port the trait-based dispatch: a `LlmClient` trait with `chat()` method, concrete implementations for Anthropic Messages API and OpenAI Chat Completions API, provider selection from config.

Implementation:

```
#[async_trait]
pub trait LlmClient: Send + Sync {
    async fn chat(&amp;self, request: ChatRequest) -&gt; Result&lt;ChatResponse, LlmError&gt;;
}
```

```
struct AnthropicClient { api_key: String, model: String, client: reqwest::Client }
struct OpenAiClient { api_key: String, model: String, client: reqwest::Client }

// Config-driven selection
// LLM_PROVIDER=anthropic  LLM_MODEL=claude-sonnet-4-20250514  LLM_API_KEY=sk-...
// LLM_PROVIDER=openai     LLM_MODEL=gpt-4o                     LLM_API_KEY=sk-...
```

Both providers support tool/function calling. Tool definitions are provider-agnostic (converted at the adapter boundary).

AI Agent Tool Definitions (6+ commands)

Tool	Description
`create_objects`	Create sticky notes, shapes, or text objects
`move_objects`	Reposition objects by ID
`update_objects`	Change properties (color, text, size)
`delete_objects`	Remove objects by ID
`organize_layout`	Arrange objects in grid, cluster, or tree
`summarize_board`	Read all text, produce summary as new sticky note
`group_by_theme`	Cluster objects by semantic similarity, color-code

Other External Services

- **PostgreSQL** — Neon (managed).
- **LLM API** — Anthropic or OpenAI, config-driven.
- No Redis, no S3, no CDN. Static frontend assets served by Axum (see §6 Hosting).

Pricing

- **Anthropic:** Sonnet is $3/M input, $15/M output. Each AI command ~$0.01-0.05. Rate limit: 10 AI commands/min/board. Budget: ~$20-40/mo.
- **Fly.io:** Dedicated machine, ~$5-10/mo.
- **Neon/Supabase:** Free tier initially, Pro ($25/mo) if needed.
- **Domain/TLS:** Fly provides `*.fly.dev` with TLS for free.

Key tradeoff: Single LLM provider, no fallback. Acceptable for a demo.

Summary: The Full Stack

Layer	Choice
Backend runtime	Rust

Layer	Choice
HTTP framework	Axum (traditional route handlers + shared services)
Architecture	Standard Axum server, frame-based WS protocol. No kernel abstraction.
Wire protocol	Frames: `{ id, parent_id, syscall, status, data }` over WebSocket
Database	PostgreSQL (Neon, `us-east`)
Real-time state	In-memory HashMap, LWW (server-timestamped, per-object versioned), server-authoritative
Persistence	Debounced 1s flush to Postgres, final flush on last disconnect
Frontend	React (Vite SPA) + Konva.js (react-konva) + Zustand
Auth	GitHub OAuth, token-based sessions, WS upgrade tickets
Access	Public-by-link (anyone with board UUID can join)
AI	Multi-provider adapter (Anthropic + OpenAI), config-driven, tool calling
Hosting	Fly.io dedicated machine, `dfw` region (Dallas)
Static assets	Axum serves built SPA, gzip + cache headers
CI/CD	`fly deploy` from terminal

Total external dependencies: LLM API (Anthropic or OpenAI) + Neon Postgres + GitHub OAuth. Everything else runs in a single Rust binary on a single Fly.io VM.

Stack divergence: This project uses Rust + Axum + React instead of the Gauntlet default (Bun + Hono + htmx) because the collaborative canvas requires rich client interactivity and the Prior project provides proven Rust/Axum WebSocket patterns. See §5 for full rationale.

Phase 3: Post-Stack Refinement

12. Security Vulnerabilities

WebSocket Authentication

Token-based WS auth via upgrade tickets: client calls `POST /api/auth/ws-ticket` with Bearer token, receives a single-use ticket (30s TTL). WS connects via `GET /api/ws?ticket=<ticket>`. Server validates the ticket (exists, not expired, not already used), deletes it, and associates the connection with the ticket's `user_id`. This avoids exposing the long-lived session token in URL query parameters (which appear in server logs and browser history).

XSS on Canvas Text

Konva renders to `<canvas>`, inherently XSS-safe. Remaining vectors:

- **Text input fields** — overlay `<textarea>` is real DOM. Sanitize on save.
- **Board names/labels** — rendered in React. Default React escaping.
- Enforce max text length (10,000 chars) in the object service.

CORS

Axum tower-http CORS layer with explicit origin allowlist. Not `*`.

AI Agent Security

Rate limiting at three layers (in-memory, per-process):

Layer	Limit	Storage
Per-user AI requests	10 requests/min	`HashMap<UserId, RateLimiter>` in-memory
Total LLM API calls	20 calls/min	Global atomic counter, reset every 60s
Token budget	50k tokens/user/hour	`HashMap<UserId, TokenCounter>` in-memory

All rate limit state is in-memory, per-process. Resets on server restart. No Redis needed — single-process deployment means no distributed state.

Prompt injection defense: Wrap user input in XML tags. Tool definitions are narrow — one per board operation, no generic tools.

Dependency Choices (Rust)

- `axum` + `tokio` — battle-tested async web stack
- `sqlx` — compile-time checked SQL queries against Postgres
- `serde` / `serde_json` — serialization
- `uuid` — ID generation
- `reqwest` — HTTP client for LLM APIs
- `tower-http` — CORS, compression, tracing middleware, static file serving
- `rand` — token generation (session tokens, upgrade tickets)

---

13. File Structure & Project Organization

Single Rust crate (no workspace) plus a `client/` directory for the React frontend.

```
collaboard/
■■■ Cargo.toml                      # Single crate
■■■ .env.example
■■■ Dockerfile
■■■ fly.toml
■
■■■ src/
■   ■■■ main.rs                     # Entry: env, init state, start Axum
■   ■■■ frame.rs                    # Frame type, Status, builders (ported from Prior)
■   ■■■ state.rs                    # AppState: PgPool + shared board state + LLM client
■   ■
■   ■■■ routes/
■   ■   ■■■ mod.rs              # Axum router assembly
■   ■   ■■■ auth.rs             # /api/auth/github, /callback, /me, /logout, /ws-ticket
■   ■   ■■■ ws.rs               # /api/ws: upgrade, frame dispatch loop
■   ■
■   ■■■ services/
■   ■   ■■■ mod.rs
■   ■   ■■■ session.rs          # Session + WS ticket CRUD (Postgres-backed)
■   ■   ■■■ board.rs            # Board CRUD + in-memory state management
■   ■   ■■■ object.rs           # Board object mutations + debounced persistence
■   ■   ■■■ cursor.rs           # Ephemeral cursor broadcast
■   ■   ■■■ ai.rs               # LLM prompt -&gt; tool calls -&gt; object mutations
■   ■
■   ■■■ llm/
■   ■   ■■■ mod.rs              # LlmClient trait + provider dispatch
■   ■   ■■■ anthropic.rs        # Anthropic Messages API (ported from Prior)
■   ■   ■■■ openai.rs           # OpenAI Chat Completions API (ported from Prior)
■   ■   ■■■ tools.rs            # Tool definitions (provider-agnostic)
■   ■
■   ■■■ db/
■       ■■■ mod.rs              # Pool init
■       ■■■ migrations/         # SQL migration files (sqlx)
■
■■■ client/                     # React frontend
■   ■■■ package.json
■   ■■■ tsconfig.json
■   ■■■ biome.json
■   ■■■ vite.config.ts
■   ■■■ index.html
■   ■■■ src/
■       ■■■ main.tsx
■       ■■■ App.tsx
■       ■■■ pages/
■       ■   ■■■ LoginPage.tsx
■       ■   ■■■ DashboardPage.tsx
■       ■   ■■■ BoardPage.tsx
■       ■■■ canvas/
■       ■   ■■■ Canvas.tsx
■       ■   ■■■ StickyNote.tsx
■       ■   ■■■ Shape.tsx
■       ■   ■■■ Connector.tsx
■       ■   ■■■ SelectionManager.tsx
■       ■   ■■■ Toolbar.tsx
■       ■■■ hooks/
■       ■   ■■■ useFrameClient.ts   # WebSocket + Frame protocol
■       ■   ■■■ useBoardState.ts    # Zustand store
■       ■   ■■■ useAuth.ts
■       ■   ■■■ useAI.ts
■       ■■■ lib/
■       ■   ■■■ frame.ts            # Frame type (mirrors Rust)
■       ■   ■■■ api.ts              # REST client for /api/auth
■       ■■■ styles/
■           ■■■ global.css
■
■■■ tests/                      # Integration tests (Rust)
    ■■■ frame_test.rs
    ■■■ board_test.rs
    ■■■ object_test.rs
    ■■■ sync_test.rs
```

Shared Types

The Frame type is defined in Rust (`src/frame.rs`) and mirrored in TypeScript (`client/src/lib/frame.ts`). Keep them manually in sync — there are only ~20 lines.

```
// client/src/lib/frame.ts
export interface Frame {
  id: string;
  parent_id?: string;
  ts: number;
  syscall: string;
  status: "request" | "item" | "done" | "error" | "cancel";
  board_id?: string;
  from?: string;
  data: Record&lt;string, unknown&gt;;
}
```

14. Naming Conventions & Code Style

Rust (backend):

Category	Convention	Example
Modules	snake_case	`board/persist.rs`
Structs/Enums	PascalCase	`Frame`, `Status`, `BoardObject`
Functions	snake_case	`handle_create()`
Constants	UPPER_SNAKE_CASE	`CHANNEL_BUFFER`
Error types	PascalCase with `Error` suffix	`BoardError`, `SessionError`
Tests	`#[cfg(test)]` inline modules	`mod tests { ... }`

TypeScript (frontend):

Category	Convention	Example
React components	PascalCase `.tsx`	`StickyNote.tsx`
Hooks	camelCase, `use` prefix	`useFrameClient.ts`
Types/interfaces	PascalCase, no `I` prefix	`BoardObject`, `Frame`
Directories	kebab-case	`canvas/`

Tooling:

- Rust: `cargo fmt` + `cargo clippy` — mandatory, run before every commit
- TypeScript: Biome for lint + format, K&R style, 4-space indentation

15. Testing Strategy

Coverage target: Happy path + smoke tests for every subsystem at MVP. Expand to edge cases and error paths post-MVP.

MVP tests (built during the 24-hour sprint):

What	Type	When	Why
Frame serialization/deserialization	Unit (Rust)	Hour 5-6	Foundation — everything breaks if frames are wrong
WS dispatch routing (prefix -> handler)	Unit (Rust)	Hour 5-6	Wrong prefix silently drops frames
Session lifecycle (create, validate, expire)	Unit (Rust)	Hour 5-6	Auth boundary — wrong token = no access
Object create/update/delete	Integration (Rust)	Hour 9-10	Validates Postgres schema + business logic
Board state hydration from Postgres	Integration (Rust)	Hour 9-10	State must survive server restart
LWW conflict resolution (version-based)	Unit (Rust)	Hour 9-10	Version bugs corrupt state
Multi-client sync smoke test	Integration (Rust)	Hour 18-19	Core product promise — two clients see each other's changes
Frame client dispatch (TS)	Unit (bun:test)	Hour 18-19	Client-side syscall routing

Post-MVP tests (Days 2-7):

What	Type	Why
AI tool dispatch	Unit (Rust, mocked)	Validates tool execution without API credits
Error paths (invalid frames, bad board IDs, unauthorized)	Unit (Rust)	Robustness
Disconnect/reconnect recovery	Integration (Rust)	Resilience requirement
Canvas component rendering	Skipped	Visual — verify manually
E2E browser tests	Skipped	Playwright setup cost exceeds value

Test commands:

```
# Rust
cargo test                          # all tests
cargo test -- --test-threads=1      # serial for DB tests

# TypeScript
cd client &amp;&amp; bun test
```

---

16. Recommended Tooling & DX

Rust backend:

Tool	Purpose
`cargo watch -x run`	Auto-restart on file changes
`cargo clippy`	Lint (mandatory before commit)
`cargo fmt`	Format (mandatory before commit)
`sqlx-cli`	Postgres migrations (`sqlx migrate run`)
`tracing` + `tracing-subscriber`	Structured logging with span context
`wscat`	CLI WebSocket testing

TypeScript frontend:

Tool	Purpose
Vite	Build + HMR
Biome	Lint + format (K&R, 4-space indent)
Zustand	State management
Bun	Package manager + test runner

Frame logging:

All frames are logged to a `frames` table in Postgres (append-only, ported from Prior's `frame_db`). This provides a complete audit trail of every operation on every board. Useful for debugging sync issues, replaying state, and understanding AI agent behavior.

Dev workflow:

```
# Terminal 1: Rust server with auto-restart
cargo watch -x run

# Terminal 2: Vite dev server with proxy
cd client &amp;&amp; bun run dev
```

Vite dev server proxies `/api` to the Rust server:

```
// client/vite.config.ts
server: {
  proxy: {
    "/api": "http://localhost:3000",
  },
}
```

Debugging WebSockets:

- Chrome DevTools > Network > WS tab for frame inspection
- `wscat -c "ws://localhost:3000/api/ws?ticket=<ticket>"` for CLI testing
- Structured `tracing` spans in Rust: one span per frame, includes syscall + board_id + user_id
- Query the `frames` table directly for post-hoc debugging (`SELECT * FROM frames WHERE board_id = '...' ORDER BY seq`)

Decision Summary

Decision	Choice	Key Tradeoff
Scale	5-20 users, single server	No horizontal scaling
Budget	$100/mo ceiling (LLM + hosting + DB)	Generous for demo, most goes to LLM
Timeline	24hr MVP, 7 days total	Tests woven into MVP (3 checkpoints)
Compliance	None	Not production-grade
Stack	Rust + Axum + React+Konva (diverges from CLAUDE.md defaults)	Slower iteration than TS, but type safety + proven patterns
Wire protocol	Frames over WebSocket	Must keep Rust/TS Frame types in sync
Database	PostgreSQL (Neon)	External dependency, but proper SQL
Sync	LWW, server-authoritative, server-timestamped, per-object versioned	No CRDT, same-object conflicts go to last writer
Persistence	Debounced 1s flush, final flush on disconnect	Up to 1s data loss on crash
Frontend	React + Konva.js + Zustand	Konva ceiling ~1-2K objects
Auth	GitHub OAuth, token-based sessions, WS upgrade tickets	Users need GitHub account
Access	Public-by-link (anyone with board UUID)	No fine-grained permissions
AI	Multi-provider (Anthropic + OpenAI), config-driven	Two adapters to maintain
Rate limiting	In-memory per-process, resets on restart	No persistence across restarts
Hosting	Fly.io dedicated, `dfw`	$5-10/mo
Static assets	Axum serves built SPA with gzip + cache headers	Single deployment artifact
Reconnect	Full snapshot on rejoin (MVP), incremental replay post-MVP	No missed-frame recovery at MVP

Notes

Prior Project

Prior is a personal Rust project (private repo: `github.com/ianzepp/prior`) that serves as the architectural reference for CollabBoard's backend. It is an agentic runtime built on a frame-based protocol where every subsystem (rooms, LLM, VFS, entity store, cache) communicates through a central kernel router using structured `Frame` messages with `{ id, parent_id, syscall, status, data }`. The kernel routes frames by syscall prefix to isolated subsystems, which respond via correlated request/item/done streams. Prior includes multi-provider LLM support (Anthropic + OpenAI) behind a normalized `llm:chat` interface, multiple gateway transports (REST/WebSocket, OpenAI-compatible, IRC, Telnet), and enforces strict code hygiene (zero `unwrap()`, pedantic Clippy, ratcheted budgets for antipatterns). CollabBoard ports Prior's frame protocol, LLM adapter pattern, and Axum gateway architecture to the whiteboard domain — the wire format and status lifecycle are identical.

Gaps to Address Later / Open Questions

These are known unknowns or deferred decisions that don't block the MVP but will need answers during or after implementation.

1. **OAuth callback token delivery.** The GitHub OAuth callback is a server-side redirect. How does the session token get to the SPA client? Options: redirect to `/#/auth?token=<token>` (hash fragment, not sent to server), or render a page that posts the token into `localStorage` and redirects. Decision deferred to implementation.
2. **WS ticket cleanup.** Expired `ws_tickets` rows accumulate. Need a periodic cleanup task (e.g., `DELETE FROM ws_tickets WHERE expires_at < now()` every 5 minutes). Same for expired `sessions`. Not designed yet — could be a `tokio::spawn` background loop or triggered lazily on new ticket creation.
3. **Board lifecycle on zero connections.** When the last user disconnects, we flush dirty objects and... then what? Keep the in-memory `BoardState` around forever? Evict after a timeout? Memory is cheap for 1-5 boards, but the eviction policy isn't defined.
4. **Object locking semantics.** The syscall table lists `object:lock` and `object:unlock` but the design doesn't specify what locking means. Options: advisory lock (UI hint, no enforcement), pessimistic lock (server rejects updates from non-holder), or optimistic lock (version check only, which we already have). Likely advisory for MVP.
5. **AI agent context window management.** Sending full board state (500 objects) to the LLM on every `ai:prompt` may exceed context limits or waste tokens. Need a strategy: summarize objects, send only visible viewport, or send object list with IDs and let the agent request details via tools. Deferred to AI implementation phase (Days 4-5).
6. **Multi-board WS connections.** Can a single WS connection join multiple boards simultaneously? The current design implies one board per connection (join sends full state). If multi-board is needed (e.g., dashboard showing live thumbnails), the frame protocol supports it via `board_id` field, but the client state management isn't designed for it.
7. **Undo/redo.** Not in the MVP requirements but users will expect it. Options: client-side undo stack (reverse operations locally, send compensating frames), or server-side undo via frame log replay. Neither is designed. The append-only `frames` table makes server-side replay possible but the implementation is non-trivial.
8. **Z-index management.** Objects have a `z_index` column but there's no strategy for assignment or reordering. Bring-to-front, send-to-back, and insert-between operations need defined behavior. Likely: bring-to-front = `max(z_index) + 1`, send-to-back = `min(z_index) - 1`, with periodic rebalancing if gaps get too large.
9. **Text editing concurrency.** Two users editing the same sticky note's text simultaneously. LWW at the object level means one user's edit overwrites the other's entirely. This is a known limitation — Miro has the same behavior. CRDTs for text are explicitly out of scope, but worth noting as a known UX issue.

10. **Connector routing.** Connectors between objects need to track source/target object IDs and recalculate paths when either end moves. The `props` JSONB field can store `{ source_id, target_id, path_type }`, but the real-time update logic (connector follows dragged object) isn't designed. This is a post-MVP feature (Days 2-3).

11. **Image/file upload.** The requirements mention potential image support. No storage backend is planned (no S3, no CDN). If needed: store small images as base64 in the `props` JSONB field (ugly but works for demo), or add a Fly.io volume for file storage. Deferred entirely.

12. **Session token rotation.** Tokens are valid for 30 days with no rotation. A compromised token gives access for the full TTL. For a demo this is fine, but a production system would rotate tokens on each use or have shorter TTLs with refresh tokens.