Here's the breakdown of an initial architecture that fits your constraints (cheap, quick, declarative, algorithm-driven, extendable), and explicitly answers where each responsibility sits.

---

1. Platform Decision

Yes, Next.js is a good path — but run it in a *client-heavy, server-light* mode.

- Board, flow, and artifact logic stay in the browser to reduce server compute.
- Server focuses on persistence, collaboration, and controlled compute tasks.

For development:

- One repo, pnpm workspaces.
- Local Postgres via Docker.
- Node server co-located with Next.js during dev.
- Deployment later can split them into pods in k8s.

---

2. Roles of Each Layer

Next.js Client

- Canvas rendering of sheets and artifacts.
- Interaction, selection, snapping, layout.
- Declarative model management in memory (CRDT-backed).
- Local execution of lightweight algorithms (layout, routing, spatial index).
- Temporary artifact state before persisting.
- MDX rendering and preview.
- Mermaid diagram rendering (client-side library).

Next.js Server (API routes)

- CRUD for sheets, artifacts, users.
- Auth and session control.
- Validation of incoming mutations.
- Export endpoints (SVG/PNG/PDF/MDX).
- Metadata search and filters.
- Hooks for AI agents to inject workflows.

Dedicated Node Server (or same process in dev)

- Real-time sync server (Yjs WebSocket or custom).
- Long-running compute:

  - Graph layout for large flows.
  - Type/schema generation from artifacts.
  - AI agent orchestration (workflow suggestions, auto-placement).
- Webhooks/event bus for external integrations.

---

3. Data Model (Essentials)

Tables in Postgres

- `projects`: id, name, owner_id, created_at.
- `sheets`: id, project_id, name, config, created_at.
- `artifacts`: id, sheet_id, type, props (JSONB), geom (JSONB), style (JSONB), meta (JSONB).
- `edges`: id, sheet_id, from_artifact, to_artifact, type, props (JSONB).
- `events`: id, sheet_id, actor_id, type, payload (JSONB), ts.
- `users`: id, email, hashed_password.

Artifact Type Examples

- `"box"`: geom, fill, label.
- `"mdx"`: source, compiled_ast.
- `"mermaid"`: source, compiled_svg.
- `"tasklist"`: tasks array, completion state.

---

4. State and Flow

In-Memory State

- Client keeps a CRDT document (per sheet).
- CRDT merges remote + local changes.
- Reducers update immutable state; render system listens for diffs.

Persistence Flow

1. Client commits change to CRDT.
2. CRDT sync server broadcasts change to peers.
3. Server batches CRDT updates → Postgres as event log.
4. Snapshots generated periodically for faster load.

---

5. Routes and Controllers

REST/GraphQL (pick one; REST easier to start)

- `GET /api/projects/:id` → project meta + sheet list.
- `GET /api/sheets/:id` → latest snapshot + events.
- `POST /api/sheets` → create sheet.
- `PATCH /api/sheets/:id` → update meta/config.
- `POST /api/artifacts` → add artifact.
- `PATCH /api/artifacts/:id` → update artifact props/geom/style.
- `DELETE /api/artifacts/:id`.
- `GET /api/export/:sheet_id.svg|png|pdf|mdx`.

Realtime (WebSocket)

- `joinSheet(sheetId)` → receive state + presence.
- `update(delta)` → broadcast CRDT changes.
- `presence(cursor, selection)` → live cursors.

---

6. Rendering on Sheet

Client-Side Algorithms

- Quadtree for hit testing.
- Constraint solver (Kiwi.js) for snapping/alignment.
- A* for connector routing.
- Layout algorithms (DAG, force-directed) from lightweight libs (`elkjs`, `dagre-lite`).
- Markdown → MDX → React tree via unified pipeline.
- Mermaid parser → SVG renderer in browser.

Artifacts render in a z-indexed scene:

- Base primitives (rect, ellipse, line).
- Embeds (mdx, mermaid, image).
- Interaction layer (selection boxes, transform handles).

---

7. Storage Strategy

- **Postgres**: primary store (JSONB for flexible props).
- **S3-compatible bucket**: images, large attachments.
- **Local IndexedDB**: offline board state + assets cache.

---

8. Algorithms from pnpm Ecosystem

- **CRDT**: `yjs` + `y-websocket`.
- **Layout**: `elkjs` (WASM) or `dagre-lite` for graphs.
- **Routing**: custom A* or `pathfinding` npm for grids.
- **Spatial index**: `rbush` or `flatbush`.
- **Constraint solver**: `kiwi` (Cassowary).
- **MDX**: `@mdx-js/mdx`, `remark-*`, `rehype-*`.
- **Mermaid**: `mermaid` npm (tree-shakable).
- **Schema validation**: `zod`.
- **DB**: `drizzle-orm` with Postgres driver.

---

9. Execution Split

Computed in Client

- Interactive transforms.
- Snapping, hit-testing.
- Rendering, MDX compilation, Mermaid rendering.
- Small/medium graph layouts.

Computed in Node Server

- Large graph layouts (batch).
- Export rendering (high-res SVG/PDF).
- AI agent workflows.
- Batch type/schema generation.

---

10. Initial Roadmap

1. **Core sheet model**: CRDT doc, artifact base type, quadtree.
2. **Rendering engine**: Canvas2D with base shapes.
3. **Persistence**: Postgres schema + simple REST.
4. **Realtime**: y-websocket integration.
5. **Artifact plugins**: box, mdx, mermaid, tasklist.
6. **Export pipeline**: SVG/PNG.
7. **Extensibility hooks**: registerArtifact, registerLayout.
8. **AI hooks**: endpoint to inject CRDT updates.

This approach lets you build from scratch with clear separation of compute and storage, keeps the initial implementation cheap and fast, and positions the sheet/artifact model to extend into complex workflows without refactoring the core.