TAKE-HOME ASSIGNMENT

Real-Time Collaborative Task Board

with Conflict Resolution & Offline Support

Role: **Full-Stack Engineer**
Time Limit: **24 hours from receipt**
Estimated Effort: **8–12 hours**

1. Overview

Build a Kanban-style task board where multiple users can simultaneously create, edit, move, and reorder tasks across columns (To Do, In Progress, Done). The board must handle real-time synchronisation via WebSockets and gracefully resolve conflicts when two users act on the same task at the same time.

Why This Assignment Exists

Any AI tool can scaffold a basic Kanban board in minutes. The real engineering challenge—and what we are evaluating—is handling simultaneous edits, drag-and-drop reordering with optimistic UI, conflict resolution when two users move the same task, and maintaining consistent state across WebSocket connections. These problems require understanding of CRDTs or operational transforms, areas where AI-generated code routinely fails on first pass.

We expect you to use whatever tools you normally use (including AI), but you must be able to explain every design decision in a live walkthrough.

2. Functional Requirements

2.1 Core Features

1. Three-column Kanban board: To Do, In Progress, Done.
2. Create, edit (title + description), and delete tasks via inline UI.
3. Drag-and-drop movement of tasks between columns and reordering within a column.

4. Real-time sync: all changes propagate to every connected client within 200 ms on localhost.

5. Multi-user presence indicators (e.g., avatars or cursors showing who is viewing/editing).

2.2 Conflict Resolution

Your system must handle the following scenarios without silent data loss:

- Concurrent move + edit — User A moves Task X to “Done” while User B edits Task X’s title. Both changes must be preserved.
- Concurrent move + move — User A moves Task X to “In Progress” while User B moves Task X to “Done.” The system must deterministically resolve which move wins and notify the “losing” user.
- Concurrent reorder — User A reorders tasks in a column while User B adds a new task to the same column. Final order must be consistent across all clients.

Document your conflict resolution strategy in a DESIGN.md file (see Deliverables).

2.3 Offline Support

- If a client loses its WebSocket connection, user actions must queue locally.
- On reconnect, queued actions replay against the current server state.
- Conflicts arising from offline actions follow the same resolution strategy as online conflicts.

3. Technical Requirements

3.1 Stack

Backend	Node.js with Express or Fastify (TypeScript preferred). Raw ws or Socket.IO for WebSockets.
Frontend	React 18+ with a drag-and-drop library (e.g., dnd-kit, react-beautiful-dnd). State management of your choice.
Database	PostgreSQL (preferred) or MongoDB. Must persist across server restarts.
Testing	At least integration tests for conflict scenarios. Unit tests for ordering logic.

3.2 Architecture Constraints

- WebSocket message handling must be separated from business logic (no monolithic message handler).
- Task ordering algorithm must be O(1) amortized per move operation (fractional indexing, linked lists, or similar). Naive array re-indexing that is O(n) per move is not acceptable.
- Database writes must be atomic—no partial updates if the server crashes mid-operation.
- The client must implement optimistic UI: the UI responds instantly to user actions, then reconciles with the authoritative server state.

3.3 Error Handling

- Graceful degradation on WebSocket disconnect (read-only mode or queued writes with visual indicator).
- React error boundaries around drag-and-drop and WebSocket-dependent components.
- Server-side validation of all incoming task mutations (no trusting the client).

4. Deliverables

1. GitHub repository (private, invite us as collaborators) with clean commit history showing your progression.
2. README.md — Setup instructions (must work with a single docker compose up or equivalent).
3. DESIGN.md — A 1–2 page document explaining your conflict resolution strategy, ordering approach, and any trade-offs you made.
4. A short screen recording (2–3 min) demonstrating: two browser tabs making simultaneous edits, a network disconnect + reconnect scenario, and the final consistent state.

5. Deployment

Your submission must include a live, publicly accessible deployment. We will use this during the review to run our live tests (Section 7.1) without needing to set up your project locally.

5.1 Free-Tier Platform Options

Deploy using any free-tier platform. Recommended options:

Platform	Best For	Notes
Railway	Full-stack deploy (backend + DB + frontend)	Free trial tier includes PostgreSQL. Supports WebSockets natively. One-click deploy from GitHub.
Lovable	Rapid frontend prototyping and hosting	AI-assisted frontend builder with instant preview URLs. Pair with Railway or Render for the backend.
Render	Backend + managed PostgreSQL	Free tier includes WebSocket support and a managed Postgres instance. Good Railway alternative.
Vercel / Netlify	Frontend static hosting	Free tier for React apps. Use alongside a separate backend host (Railway / Render) since these do not support persistent WebSockets.

5.2 Deployment Requirements

1. The live URL must be included in your README.md under a “Deployment” section.
2. WebSocket connections must work on the deployed version (not just localhost).
3. The deployed app must connect to a persistent database (not in-memory).
4. The deployment should survive a cold start—free tiers often spin down after inactivity. Document expected cold-start time if applicable.
5. If your architecture splits frontend and backend across platforms (e.g., Lovable + Railway), document both URLs and the connection flow.

Cost Reminder

Do not spend any money on this assignment. All recommended platforms offer free tiers sufficient for this project. If a platform requires a credit card for sign-up, you may use an alternative from the list above.

6. Evaluation Rubric

Each criterion is scored 1–5. The weight column indicates relative importance.

Criterion	Weight	Excellent (5)	Acceptable (3)	Poor (1)
Conflict Resolution	**25%**	All 3 scenarios handled; no data loss	Most conflicts handled; minor edge cases	Silent data loss or crashes
Real-Time Sync	**20%**	< 200 ms propagation; consistent ordering	Works but occasional flicker or delay	Requires refresh to see changes
Code Architecture	**15%**	Clean separation; testable modules; typed	Reasonable structure; some coupling	Monolithic; untestable
Offline Support	**15%**	Full queue + replay with conflict handling	Basic queue; some replay issues	No offline handling
Deployment	**10%**	Live URL works; WebSockets stable; DB persists	Deployed but minor issues (cold start, flaky WS)	Not deployed or broken URL
Testing	**10%**	Integration tests for all conflict cases	Some tests present	No meaningful tests
Documentation	**5%**	Clear DESIGN.md; easy setup; deploy docs	Partial docs; setup works	Missing or misleading docs

7. Candidate FAQ

Can I use AI tools? Yes. Use whatever tools you normally use in your workflow. However, you must be able to explain every line of code and every design decision during the live walkthrough. If you cannot, it will count against you.

Can I use a different stack? The backend and database are flexible.

What if I run out of time? Prioritise conflict resolution and real-time sync over polish. A working system with rough UI that handles conflicts correctly will score higher than a beautiful board that silently drops changes and deployment

How realistic should the offline support be? We expect a working queue-and-replay mechanism, not a full offline-first database. IndexedDB or an in-memory queue is fine.

Do I need authentication? No. Simple user identification (e.g., a randomly generated username on connect) is sufficient.