Project Documentation Essays

1. Architecture Choices

This Kanban board application follows a layered architecture pattern using React Context API with `useReducer` for centralized state management. The architecture separates concerns into four main layers: components, context, hooks, and services.

State Management: The application uses a split context pattern implemented in `BoardContext.js`, providing separate contexts for state (`BoardStateContext`) and dispatch (`BoardDispatchContext`). This separation prevents unnecessary re-renders—components that only dispatch actions do not subscribe to state changes, improving performance significantly.

Reducer Pattern: All state transitions are handled through `boardReducer.js`, which processes actions defined in `boardActions.js`. The reducer handles board operations (load, reset), column operations (add, rename, archive), card operations (add, update, delete, move), and UI state (modals, dialogs).

Service Layer: The `services/` directory contains business logic separated from React components. The `syncQueue.js` implements a singleton pattern for managing offline sync operations, while `conflictResolver.js` handles three-way merge logic. This separation enables unit testing without React rendering overhead.

Provider Composition: `BoardProvider.jsx` manages local state and persistence, while `SyncProvider.jsx` handles network synchronization. This composition allows independent testing and the flexibility to disable sync features without affecting core board functionality.

2. Optimistic Updates Workflow

The application implements optimistic updates to provide immediate UI feedback while synchronizing changes with the server in the background. This pattern significantly improves perceived performance and enables offline functionality.

Immediate State Update: When a user performs an action (e.g., moving a card), the `boardReducer.js` processes the action immediately, updating the UI without waiting for server confirmation. The action is dispatched through `BoardProvider.jsx`, which applies the change to local state instantly.

Queue Management: Simultaneously, syncable actions (defined in `SYNCABLE_ACTIONS`) are enqueued in `syncQueue.js` with the current board version. The queue persists to local Storage, ensuring pending changes survive page refreshes. Each queue item contains the action payload, timestamp, retry count, and version number.

Background Synchronization: The `SyncProvider.jsx` processes the queue when online, sending actions to the server via the API service. A background interval (`SYNC_INTERVAL = 30000ms`) periodically processes pending items. When the browser comes back online, an event listener triggers immediate queue processing.

Error Handling and Revert: If synchronization fails, the system implements exponential backoff with configurable retry limits (`MAX_RETRIES = 3`). The `SyncStatus.jsx` component displays pending count and allows users to manually retry or clear failed items. Version conflicts trigger the conflict resolution flow rather than automatic reversion.

Visual Feedback: The UI displays sync status through `SyncStatus.jsx`, showing online/offline state, pending action count, and last sync timestamp.

3. Conflict Resolution Approach

The application implements three-way merge conflict resolution to handle concurrent edits from multiple clients or browser tabs. This approach compares local changes, server changes, and a common base state to identify and resolve conflicts.

Conflict Detection: The `detectConflicts()` function in `conflictResolver.js` analyzes differences between three states: base (last known synchronized state), local (current client state), and server (latest server state). It uses `getDiff()` to identify changes from base in both local and server versions.

Conflict Types: The system recognizes several conflict categories:

- `SAME_FIELD`: Both clients modified the same property (e.g., card title)
- `DELETE_MODIFY`: One client deleted an item the other modified
- `MOVE_CONFLICT`: Same card moved to different destinations
- `ORDER_CONFLICT`: Column ordering changed differently

Resolution Strategies: The `MergeStrategy` defines available approaches: `AUTO_MERGE` for non-conflicting changes, `LOCAL_WINS` to prefer client state, `SERVER_WINS` to prefer server state, and `MANUAL` for user intervention. Users can select `ResolutionChoice` options: keep local, keep server, keep both, or provide custom resolution.

User Interface: `ConflictResolutionModal.jsx` presents conflicts visually, displaying local versus server values for each conflicting field. Users resolve conflicts individually, and `applyResolutions()` constructs the final merged state based on their choices.

Merge Execution: After resolution, the merged state is sent to the server via `/api/board/merge` endpoint, which validates the merge against current server version before accepting.

4. Performance Optimizations

The app implements several performance optimizations to maintain responsiveness with large datasets and frequent user interactions.

Context Splitting: Separating `BoardStateContext` and `BoardDispatchContext` prevents components that only dispatch actions from re-rendering when state changes. This optimization is particularly effective for action buttons and form handlers.

Callback Memoization: Components use `useCallback` and `useMemo` hooks to prevent unnecessary function recreation. In `ListColumn.jsx` and `Card.jsx`, event handlers are memorized with appropriate dependencies, preventing child component re-renders during parent updates.

List Virtualization: `VirtualizedCardList.jsx` implements windowed rendering using `react-window` library. When card count exceeds `VIRTUALIZATION_THRESHOLD = 30`, only visible cards plus an overscan buffer render to the DOM. This reduces initial render time and memory usage for columns with many cards.

Debounced Persistence: `BoardProvider.jsx` implements debounced local Storage writes with `SAVE_DEBOUNCE_MS = 500`. Rapid state changes (e.g., during drag operations) batch into single storage operations, preventing main thread blocking.

Code Splitting: Modal components (`CardDetailModal`, `ConfirmDialog`, `ConflictResolutionModal`) use `React.lazy()` for code splitting. These components load on-demand when first accessed, reducing initial bundle size. Corresponding fallback components in `LoadingFallback.jsx` display during loading.

Efficient Re-renders: The `memo()` wrapper on `Card` and `CardRow` components enables shallow prop comparison, preventing re-renders when props remain unchanged.

5. Accessibility Implementation

The app implements WCAG 2.1 Level AA accessibility guidelines through semantic HTML, ARIA attributes, keyboard navigation, and screen reader support.

Focus Management: The `useFocusTrap` hook in `useAccessibility.jsx` constrains keyboard focus within modal dialogs. It handles Tab and Shift+Tab cycling, traps focus on first/last elements, and restores focus to the triggering element on close. Modals include `role="dialog"`, `aria-modal="true"`, and `aria-labelledby` attributes.

Keyboard Navigation: The `useArrowNavigation` hook enables arrow key navigation within card lists and columns. All interactive elements are reachable via Tab key, and `@dnd-kit` provides keyboard-accessible drag-and-drop through Space/Enter activation and arrow key movement.

Screen Reader Support: The `useAnnounce` hook provides live region announcements for dynamic content changes. Drag operations announce pickup, movement, and drop events. The `VisuallyHidden` component adds screen-reader-only labels for icon buttons without visible text.

Semantic Structure: The application uses semantic HTML elements: `<header>` for the app header, `<main>` for board content, `<section>` for columns, and `<article>` for cards. Heading hierarchy follows logical order (h1 for app title, h2 for column titles).

Form Accessibility: Form inputs include associated `<label>` elements, `aria-required` for mandatory fields, and `aria-invalid` with `aria-describedby` for validation errors. The `SkipLink` component allows keyboard users to bypass navigation.

Validation: Accessibility testing used axe DevTools browser extension across all application states, achieving zero critical or serious violations.