ChatGPT

Front-End Module Implementation Report

1. Architecture Spec

- **Rendering Model (RSC + Client Islands):** The app uses Next.js App Router with React Server Components (RSC) for static content and Client Components for interactive UI. The overall page (including Header, Footer, and static parts of the dashboard) is rendered on the server for performance, while dynamic widgets and the Sidebar are **client components** (declared with `"use client"`). This allows heavy lifting (data fetching, initial markup) to occur server-side, sending minimal JS to the browser [1]. Interactive parts (drag/drop grid, context menus, AI chat) hydrate as isolated client-side islands, interleaving with server-rendered UI. The Header/Footer can be server components (if mostly static), whereas each dashboard **Section** and the Sidebar (with its toggle logic) are client components within the RSC tree.

- **State Management (Zustand Store):** We use a centralized Zustand store for UI state that spans components (layout configuration, which sections are hidden, sidebar open state, etc). Zustand provides a lightweight observable state container. The `useDashboardStore` holds an array of section widgets with their layout (position, size) and visibility, as well as global flags (e.g. `isSidebarOpen`, `sidebarMode`). This global store allows any component (e.g. a Section or the Grid) to read/update state without prop-drilling. The store also manages optimistic updates for offline changes. (Jotai could be an alternative for fine-grained atomic state, but Zustand's simpler API fits well for a shared dashboard state.)

- **Data Flow (React Query + RSC):** Data for each widget (financial metrics, charts, etc.) is loaded via React Query hooks in client components. Wherever possible, data is also fetched in server components (using Next.js RSC) so that initial HTML includes critical data (SSR for first load). We integrate **React Query** for caching and background re-fetching of data from the "intelligence layer" APIs. Each widget component uses a React Query hook to retrieve its data (e.g. portfolio summary, market data) as a **read model**. The query results are cached and shared, enabling widgets to get updated data when network is available. On first load, Next.js can fetch these read models in a server route (or directly in RSC) to avoid loading spinners. Subsequent interactions rely on React Query's client cache (with stale-while-revalidate updates). The AI agent in the Sidebar also uses this data (via the intelligence layer APIs) to answer user queries or generate proposals.

- **Offline Strategy:** The app is offline-ready. We leverage **IndexedDB** (via the `idb` library) to store cached data and pending changes. React Query's persistence is configured so query results (read data) are saved to IndexedDB, enabling the dashboard to display last-known data when offline [2] [3]. For writes, we implement an offline **write queue**: user actions (e.g. reordering a widget, accepting an AI proposal) are recorded as change objects in IndexedDB ("outbox"). A network connectivity listener (or React Query's offline mutation support) detects when connection is restored and then flushes the queue, applying each change via API calls in order. We use an exponential backoff for retries if the network remains unavailable. This strategy, combined with service worker

caching for static assets (via Next.js PWA support), allows the app to function up to several hours offline and seamlessly sync when reconnected.

2. Component Contracts (TypeScript Interfaces)

We define clear TypeScript interfaces for core components and data structures:

```
// Unique identifier types for widgets and sections
type SectionId = string;
type WidgetType = 'chart' | 'table' | 'metric' | 'news' | ...;

// Dashboard section (widget) data model
interface SectionConfig {
  id: SectionId;
  type: WidgetType;
  title: string;
  position: { x: number; y: number };  // grid coordinates (columns/rows)
  size: { w: number; h: number };      // width/height in grid units
  visible: boolean;
}

// Props for the overall Grid layout component
interface GridProps {
  sections:
SectionConfig[];                 // all section configs (visible and hidden)
  onLayoutChange?: (updated: SectionConfig[]) => void;
}

// Each SectionContainer wraps a widget in the grid
interface SectionContainerProps {
  config: SectionConfig;
  children: React.ReactNode;                // the widget's content (chart,
etc.)
  onResize?: (id: SectionId, newSize: SectionConfig['size']) => void;
  onDragEnd?: (id: SectionId, newPos: SectionConfig['position']) => void;
  onHide?: (id: SectionId) => void;
  onOpenContextMenu?: (id: SectionId, anchor: {x:number, y:number}) => void;
}

// Sidebar with AI agent (two modes: overlay or push)
interface SidebarProps {
  mode: 'overlay' | 'push';
  open: boolean;
  onToggle: () => void;
  width?: string | number;    // width in px or %, used for push mode layout
}
```

```ts
// The Sidebar may contain an AI Agent component
interface SidebarAgentProps {
  onSubmitQuery: (query: string) => void;
  proposals: AIProposal[];    // list of AI-generated proposals for user review
  onApproveProposal: (pid: string) => void;
  onRejectProposal: (pid: string) => void;
}

// Context menu for section widgets
interface ContextMenuProps {
  targetSectionId: SectionId;
  items: ContextMenuItem[];
  onClose: () => void;
}
interface ContextMenuItem {
  label: string;
  shortcut?: string;
  action: () => void;
  dangerous?: boolean;   // e.g., delete/remove action styling
}

// Example: AI-generated proposal/change-set structure
interface AIProposal {
  id: string;
  description: string;    // e.g. "Reorder widgets for optimal view"
  changes: SectionConfig[];  // proposed new layout or config changes
  state: 'pending' | 'accepted' | 'rejected';
}
```

These interfaces define contracts: for example, `SectionContainer` receives a `SectionConfig` (with all info needed to render and position the widget) and callbacks for user actions (resize, drag, hide). The `SidebarProps` allow toggling and switching modes. The `SidebarAgentProps` defines how the AI agent and proposal system communicates (a list of proposals with handlers to approve/reject). The `ContextMenuProps` carries menu items (each with label, optional keyboard shortcut, and action handler) for widget-level actions like Refresh data, Hide widget, Settings, etc.

3. Layout & Grid Spec

- **Numeric Grid Setup:** The dashboard's central area is a responsive CSS grid with a 12-column layout (representing 12 sections). We partition the screen into a 12-column by N-row grid; each widget section occupies some subset of this grid. By default, we display 12 sections (each widget initially sized to occupy one column width or more as designed). The grid uses Tailwind CSS utility classes (e.g. `grid-cols-12`) to define the column structure. Each **SectionContainer** is absolutely positioned in this grid using CSS grid line placements or with a library that manages grid positioning. We use numeric coordinates `(x, y, w, h)` in the SectionConfig: for example, a widget might start at column x=0, row y=0, spanning w=3 columns and h=2 rows. This numeric layout approach

enables deterministic placement and easy collision detection (similar to how libraries like React-Grid-Layout define layouts). All sections are contained in a parent `<div class="grid grid-cols-12 auto-rows-min gap-4">` (with Tailwind classes for spacing).

- **Responsive Breakpoints:** The grid reflows at defined breakpoints to ensure usability on smaller screens. We define breakpoints (matching Tailwind's defaults or custom): e.g. at `xl` (≥1280px) we use 12 columns, at `lg` (≥1024px) maybe 8 columns, at `md` (≥768px) 6 columns, and at `sm` or below a single-column stack. Each SectionConfig can have a layout per breakpoint (if needed) or we can auto-collapse sections into the next line as columns reduce. For simplicity, on mobile (small screens) all sections stack vertically (single column). The layout persistence (see below) stores the last known arrangement for each breakpoint so that user customizations persist across screen sizes.

- **Persistence Schema:** We persist the dashboard layout (positions, sizes, visibility of each of the 12 sections) in local storage or IndexedDB so that users' customizations are not lost on refresh. The schema can be a JSON array of SectionConfig objects. For example, in IndexedDB we maintain a `layouts` store keyed by user or dashboard ID, containing the array of sections. On load, the app checks for a saved layout; if none, a default 12-section layout is loaded (likely from a static JSON or the server). Any changes (drag, resize, hide/show) update this local schema (with debouncing) and also queue an update to the server (so the layout persists across devices after sync). Versioning of this schema is handled via a `version` field so we can migrate if the layout model changes.

- **Collision Rules:** The grid uses a **no-overlap** rule for widgets. Only one section can occupy a given grid cell at a time. When dragging or resizing a section, we detect potential collisions with others:

- If a widget is dragged to a position that overlaps an existing one, we **push** the obstructed widget downwards (or to the right) to the next free space (a behavior similar to packery or masonry layouts). The movement is animated (with FLIP, see Motion Spec) so widgets smoothly shuffle.
- If the user tries to resize a widget larger and it would collide, we either prevent the resize beyond the collision or simultaneously push/move the other widgets if space permits.
- The layout algorithm ensures widgets always snap to grid coordinates. During drag, a semi-transparent preview of the widget's new position is shown snapping to the grid. We use an **incremental collision detection**: as the widget moves, we continuously check the target grid cells. We can leverage an existing library (like `@dnd-kit` with collision detection sensors) to simplify this, or implement manually by checking the SectionConfig array for any with overlapping x/y ranges.
- Hidden sections (visible=false) are removed from layout calculations (their space is freed).
- The grid honors a max row limit or scroll: if many widgets are pushed out of initial view, the container can scroll vertically.

4. Motion Spec

- **Motion Tokens:** All animations use a consistent set of durations and easing curves defined as design tokens. For example, we establish tokens like `--motion-duration-short: 100ms`, `--motion-duration-medium: 300ms`, `--motion-duration-long: 500ms` and easing presets like `--motion-ease-soft: cubic-bezier(0.4, 0.0, 0.2, 1)` (ease-in-out) and spring parameters for overshoot effects. These tokens are defined in a central file (and reflected in Tailwind config if needed). By using tokens, we ensure timing consistency across the app (e.g., the sidebar toggle and

a widget resize both feel cohesive). We also include a reduced-motion token that is 0ms or instant for users who prefer no animation.

- **FLIP Transitions (Reordering and Resizing):** We implement **FLIP** (First-Last-Invert-Play) animations for moving sections in the grid. Using Framer Motion, we add the `layout` prop to each motion-enabled SectionContainer, which automatically computes the element's previous and next position and animates the difference. When the user drags a widget to a new spot or widgets reorder, Framer Motion will smoothly transition them to their new positions using transforms, avoiding jarring re-layout jumps. If needed, we manually implement FLIP: capture a widget's initial position (First), on layout change get its new position (Last), apply an inverted transform to keep it visually in place, then animate that transform to zero (Play). This technique makes drag-and-drop reordering look fluid. Resizing a widget similarly uses FLIP: the content scales or reflows during the resize with an animation, and affected neighboring widgets slide into new positions.

- **Bar↔Line Chart Morph:** Some widgets (charts) can toggle between bar and line representations. We enable a smooth morphing animation between these states. Using D3 and visx, we represent the chart shapes (bars and lines) as SVG path data. When toggling, we compute an interpolated path between the bar shape and the line curve. For example, bars can be converted to a series of points and a continuous path; using d3's shape interpolators or a library like Flubber, we tween the SVG path `d` attribute from the bar outline to the line path over, say, 400ms. This creates a visually pleasing morph where bars "melt" into a line or vice versa. Data values remain constant during the transition (we're just changing representation). We also animate other properties: the fill of bars might fade out to the line's stroke color. The transition uses an ease-in-out curve to emphasize the morph. If the morph is too complex on older devices, a fallback could cross-fade between a bar chart and a line chart component.

- **Sidebar Mode Animation:** The Sidebar supports **push** and **overlay** modes, each with distinct animations:

- In **overlay mode**, opening the sidebar slides it in from the left over the content. We use a translation animation: the sidebar's `<aside>` starts at `transform: translateX(-100%)` (off-canvas) and animates to `translateX(0)`. Meanwhile, the main content can remain stationary (with perhaps an optional dimmed overlay to focus the sidebar). Closing the sidebar slides it back out to the left.
- In **push mode**, the sidebar is part of the layout flow. Opening it causes the main content area to shrink or move to the right. We animate this by adjusting the content container's margin or transform. For example, when sidebar opens, we animate the main content container's margin-left from `0` to the sidebar's width (or use transform translateX). This push is accompanied by the sidebar panel fading in or simply appearing. The transition should be synchronous, so the sidebar and content move together.
- Switching the sidebar **mode** (push ↔ overlay) dynamically will also be animated. For instance, if the sidebar is open in push mode and the user switches to overlay, we might smoothly layer it on top: increasing its z-index and transitioning the content's margin back to zero while the sidebar stays visible, giving a seamless mode change.

- All sidebar animations use a medium-duration token (~300ms) and an easing that feels smooth. We also trap focus (for accessibility) when in overlay mode with a modal backdrop.

- **Reduced Motion Fallback:** If the user has `prefers-reduced-motion` enabled or a low-performance device is detected, we minimize or disable these animations. The app will detect this preference (via a media query or `window.matchMedia`) and adjust: FLIP animations will jump immediately to end positions without tweening, the chart morph will be replaced by an instant swap of chart types (or a simple fade), and the sidebar will appear/disappear without sliding. We ensure all our Framer Motion animations respect a global flag that can switch them to no-animation mode. This guarantees accessibility for motion-sensitive users and a functional experience if animations are turned off.

5. Accessibility & Input

- **APCA Contrast Targets:** We adopt **OKLCH color tokens** for theming and use APCA (Advanced Perceptual Contrast Algorithm) to ensure accessible contrast. All text and UI elements meet recommended contrast levels: for instance, body text on its background aims for an APCA lightness contrast (Lc) of around 90 (which corresponds to very high contrast for fluent reading [4]), and important UI labels meet at least Lc 60-75. We use the OKLCH color space to adjust lightness and chroma of our palette to hit these targets, verified with design tools. Tailwind is configured with OKLCH color values and we test our color combinations with APCA to ensure compliance, going beyond legacy WCAG ratios. In practice, this means using sufficiently dark text on light backgrounds (or vice versa) for all dashboard text, including small chart labels and numbers (which should hit ~Lc 60 for readability).

- **Keyboard Shortcuts:** We provide keyboard shortcuts for common actions to improve efficiency for power users. For example:

- Pressing `[` or another key could toggle the Sidebar (open/close the AI agent panel) quickly.
- When focus is on a Section widget, arrow keys can be used to navigate between sections (e.g., move focus to the next section tile in the grid), and **Enter** might open that section's expanded view or context menu.
- **Esc** closes any open overlay (sidebar in overlay mode, context menu, or modal).
- In the context menu, users can press the underlined letter or arrow keys + Enter to activate menu items.

- We ensure no shortcut conflicts with browser defaults or assistive tech (e.g., we avoid single-letter keys that conflict with screen reader navigation, and allow customization if needed). All shortcuts are documented in a keyboard help sheet (maybe opened by pressing `?`).

- **Focus Order & Management:** The UI is structured with a logical tab order: Header elements (e.g. navigation, profile menu) come first in the DOM and tab sequence, followed by the Sidebar toggle (even if hidden visually, it's accessible to screen readers to open the agent), then the grid of sections (each section container has a tabindex or contains focusable content). Within each Section widget, there may be interactive controls (like a "more options" button that opens the context menu). We manage focus such that:

- When the Sidebar opens in overlay mode, focus is moved into the sidebar (to the chat input of the AI agent by default). The sidebar in overlay acts as a **focus trap** – cycling within the agent's UI – until closed (to prevent focus going behind it).

- When a context menu is opened for a widget (via right-click or keyboard), focus goes into the menu and returns to the originating widget after closing.
- The focus order of sections follows a row-major order (left-to-right, top-to-bottom in the grid) so that tabbing moves through widgets in a predictable way.

- All interactive elements have clear focus styles (using Tailwind's focus-visible ring with colors that meet contrast).

- **Escape Key Policy:** The **Esc** key is globally captured to handle common dismiss actions. Pressing Esc will:

- Close the Sidebar if it's in overlay mode and currently open (and remove its focus trap so focus returns to the toggle or the element that opened it).
- Dismiss any open context menu or tooltip.
- Cancel any modal dialog or popover if we introduce any in the future.

- We ensure Esc does not unintentionally close something that the user didn't intend (we check the highest priority context: if a modal is open, Esc closes that first, else if sidebar open then close it, etc.). This behavior is communicated to users (e.g., via ARIA labels or user guide).

- **Screen Reader Labels & ARIA:** All components include appropriate ARIA attributes and labels to be usable by screen readers:

- The Sidebar toggle button has `aria-label="Open AI Assistant Panel"` (toggling to "Close..." when open).
- The Sidebar itself is marked with `role="complementary"` and labelled (aria-labelledby) by its header (e.g., "AI Assistant").
- Each SectionContainer has an `aria-label` that reflects its title (e.g., "Portfolio Performance chart, drag to move" and we update `aria-live` region or announce when its position changes after a drag, like "Moved Portfolio Performance to column 2, row 1").
- Draggable elements have `aria-grabbed` attributes toggled during drag as needed, and the drop targets (the grid cells) might be given `aria-dropeffect` or we announce via live region that a drop is successful.
- The ContextMenu is implemented with proper menu semantics: `role="menu"` on the container, `role="menuitem"` on each item, and keyboard support (Up/Down to navigate, with the first item focused on open). We ensure menu items have clear text labels. If an item has a keyboard shortcut, we include that in its accessible name (e.g., "Hide Widget, shortcut H").
- For charts and other visual content, we include off-screen descriptions or summaries (e.g., a brief `<p class="sr-only">Line chart showing revenue over time, rising from Q1 to Q4.</p>` within the widget) so that screen reader users get an overview of the data or a way to request an accessible data table.
- All interactive controls meet at least WCAG 2.1 AA standards (focusable, operable by keyboard, with visible focus indicators, etc.). We test using axe and manual screen reader navigation to ensure a smooth experience.

6. Autosave & Versioning

- **Debounced Autosave:** User edits to the dashboard (moving or resizing a section, toggling the sidebar mode, editing widget settings, etc.) are auto-saved. Rather than requiring a manual "Save" button, the app debounces persistence operations to avoid overwhelming storage or network. For example, if the user drags several widgets in quick succession, we wait say 1-2 seconds after the last drag to save the new layout. This debounce ensures we capture final state, not intermediate thrashes, and group related changes together.

- **Local Draft Mirror (IndexedDB):** Every change is first written to a **local draft** in IndexedDB. We maintain an object store for the current “DashboardDraft”. On any change, we update this draft (overwrite the current layout state) *immediately* so that even if the user closes the tab or the app crashes, the latest state is safely kept client-side. This local draft acts as a source of truth when offline. It includes the layout schema and perhaps a timestamp or version.

- **Server Sync After Idle:** The server is updated after a period of user inactivity or low frequency changes. We use an idle timer (e.g., 5 seconds of no changes) to trigger a sync. When triggered, a background process takes the accumulated changes (since last sync) and sends them to the server via an API call (for example, a PUT request with the latest layout and a list of changes). We could also implement a **commit log**: each atomic change (e.g., "section X moved from col1 to col2") is sent as an entry. However, sending the final state is simpler and reduces version conflicts. The server can optionally store a version history of layouts.

- **Change-Set Ledger:** For auditing and possible rollback, we maintain a **ledger of change-sets**. Each user action produces a change-set object (with an ID, timestamp, user ID, and details of the change). This is stored in IndexedDB as well, possibly in an append-only log store. For example, moving a widget produces `{id: 'chg123', type: 'move', sectionId: 'portfolio', from: {x:0,y:0}, to: {x:4,y:0}, timestamp: ...}`. The ledger allows us to replay changes if needed (helpful for debugging or syncing) and could be used to implement undo/redo in future. When syncing to the server, we could send the entire change-set log or just the final state; but we keep the log locally regardless. If the server also keeps versions, we tag each change-set with a version number to correlate. After a successful server sync, we mark those change-sets as synced (or remove them if we only need server history). In case of conflicts (e.g., if the server has a newer version than the client’s base), the ledger can help in merging or diagnosing, though in our single-user scenario we expect minimal conflict.

7. Offline Mode

- **Cache Read Models:** The app’s data layer uses a cache-first strategy so that most data is available offline after it’s loaded once. We use React Query’s **persisted cache** to store API responses (the read models from the intelligence layer) in IndexedDB [5] [2]. This means if the user loses connectivity, the dashboard widgets still have access to the last fetched data (e.g., yesterday’s stock prices, last known account balance). The widgets will display cached data with an “offline” indicator rather than failing. For implementation, we wrap the app in React Query’s `<PersistQueryClientProvider>` with an IndexedDB persister, adjusting cache time-to-live to a long duration so data isn’t garbage-collected prematurely. Additionally, certain static data (like list of widget types, user preferences,

design tokens) can be preloaded and stored in localStorage or as part of the app bundle for offline use.

- **Write Queue Format:** All user modifications made offline are queued for delivery when online. We structure this as an **Outbox** in IndexedDB. Each entry in the outbox is a small JSON representing an action to sync, e.g. `{id: 'offline123', action: 'moveSection', sectionId: 'revenueChart', newPos: {x:2,y:0}, timestamp: 1695850000000}`. The queue preserves order of actions. We also record dependencies if any (e.g., if a proposal acceptance must happen before a particular layout change, though in our case changes are mostly independent). When online, the app will iterate through this queue and attempt each action via the appropriate API (for instance, call `PATCH /layout` or a specific endpoint). If an action fails (network or server error), it stays in the queue and we'll retry later. We also mark each with a retry count and last attempted time to implement backoff.

- **Sync Badge Indicator:** The UI includes an indicator of offline status and unsynced changes. For example, in the Header or Sidebar, we have an icon (like a cloud slash or a dot) that turns red when offline. If there are pending sync items in the outbox, a badge shows the count (e.g., "3 changes pending"). On reconnect, when sync begins, we might change the icon to a syncing state (rotating arrow). Once all changes are synced successfully, the badge clears. This gives users transparency that their actions are saved and will be uploaded. The Sidebar's AI agent might also indicate if it's offline (since it might not be able to fetch answers while offline, we show a message there).

- **Retry & Backoff Policy:** The app listens for the browser `online` event to trigger sync immediately when connectivity returns. Additionally, if the app stays open during offline, we periodically re-attempt sync in the background (say every 30 seconds, with a backoff doubling each time up to a few minutes). Our backoff strategy: first retry 5 seconds after a failure, then 15s, 30s, 60s, etc., up to perhaps 5 minutes max interval. If a sync fails due to a server error (HTTP 500 or specific error response), we might pause and require manual intervention (like a refresh or sign-out if auth failed). The user will be informed if a change cannot be synced (e.g., an error icon next to the pending change, with an option to retry now or discard it). However, typically network outages are temporary, and the queue will drain automatically. We ensure that partial syncs handle ordering: e.g., if the first item in queue fails, we stop and don't apply subsequent changes out of order (to avoid dependency issues). The entire offline mechanism is thoroughly tested to avoid data loss or duplication (see Testing Plan).

8. Integration Points

- **Design Tokens & Theming:** The front-end module consumes a central design system (color, typography, spacing tokens) likely defined by the product's style guide. We integrate TailwindCSS with custom tokens by extending the config – for example, injecting an **OKLCH color palette** that matches the brand and accessibility requirements. We use CSS variables for colors, which Tailwind can reference, enabling dynamic theming (e.g., light/dark mode if needed, or high-contrast mode). Spacing and sizing tokens (for consistent padding, margin, etc.) are also configured. The layout and components use these tokens via Tailwind utility classes (e.g., `text-primary`, `bg-elevated`, `p-4` for spacing). This module doesn't directly decide colors, but it ensures to use the design tokens classes provided (for instance, a `token.ts` file might export color values and we ensure any raw CSS in charts uses those). If the design tokens include motion and radius, we use them too (e.g.,

border-radius tokens for widget containers). **APCA compliance** is checked as part of design token creation (see Accessibility above), and our use of tokens ensures those contrast-tested values propagate throughout.

- **Intelligence Layer (Read Models):** The financial app likely has an "intelligence layer" that provides processed data (read models) for the front-end, possibly via an API or as part of the Next.js backend. Our front-end module interfaces with this layer through **React Query** calls. For example, there might be an endpoint `/api/dashboard/sections` that returns all 12 sections' data. On page load, a server component could fetch this and pass it to client components. Alternatively, each widget calls its specific endpoint (e.g., a chart calls `/api/data/portfolio-performance`). The integration is such that if the intelligence layer updates (like new AI insights or analytics), those appear in the UI via the normal data fetch cycle (React Query will refetch on interval or when triggered by user). We also handle real-time updates if applicable: perhaps via WebSockets or SSE if the intelligence layer pushes events (for instance, a stock price update or an AI alert). In that case, the front-end subscribes and updates the zustand store or invalidates queries. The **AI agent** in the Sidebar is also a client of the intelligence layer: when a user asks a question or requests an analysis, the agent component calls an AI API (could be OpenAI or a custom model) possibly hosted in the intelligence layer. The question, along with relevant context (user data, selected widget info) is sent, and the answer or proposal is returned and displayed.

- **AI Proposals & Approvals Workflow:** A key integration is between the UI state and the AI assistant's proposal system. The Sidebar's AI agent can generate **proposals** – for example, "I noticed your budget widget is hidden, would you like to unhide it?" or "Reorder the sections to prioritize KPIs at top?". The integration design:

- The AI agent, upon computing a suggestion, will create an `AIProposal` entry (as defined in our interface) and add it to a Zustand store or local state within the agent. This triggers the UI to display a notification or list of proposals in the sidebar.
- Each proposal includes a description for the user and the actual changes (for example, a list of SectionConfig modifications or an action like "add new section X").
- The user can click "Apply" (approve) or "Dismiss" (reject). On **approval**, the front-end will take the proposal's changes and apply them just as if the user made those changes manually: e.g., update the dashboard store layout state, update the UI immediately (optimistically), and add the change-set to the offline queue or sync directly if online. The proposal's state flips to 'accepted' and it might be archived or removed from the list. If **rejected**, we mark it 'rejected' and do nothing else (or possibly provide feedback to the AI for learning, if applicable).
- We ensure that applying a proposal goes through the same autosave & versioning flow – meaning it will be persisted and synced exactly like a user drag/drop action. This keeps the system consistent (the server doesn't care if a change came from AI or user, but we might tag it in telemetry).
- From a UI perspective, proposals might be shown as a list in the Sidebar with each having an "Approve" and "Reject" button (which are keyboard accessible and have clear labels like "Press A to apply suggestion").
- Security/Validation: We treat AI proposals carefully – they are essentially untrusted until approved. The integration layer might enforce that proposals are only within certain bounds (e.g., the AI cannot delete all widgets without user confirmation, obviously). On the front-end, we double-check that the changes from a proposal are valid (e.g., section IDs exist, positions are within grid, etc.) before applying.

- This integration ensures the AI is a helpful assistant that can make changes, but the user remains in control, satisfying the “proposals/approvals” requirement.

9. Performance & Telemetry

- **Animation Performance Targets:** All interactive animations (dragging, resizing, chart morphs) are optimized to maintain a 60fps frame rate on modern devices. We aim to budget no more than ~16ms of work per frame. We achieve this by offloading expensive calculations outside the animation loop:
- Layout thrashing is minimized by using transforms (translate/scale) for movement instead of setting absolute positions during the animation, which allows the browser’s compositor to handle it cheaply.
- For drag, we utilize `requestAnimationFrame` loops and only update positions then, or rely on Framer Motion’s internal optimization.
- We avoid shadow or filter effects that would slow down compositing, or use them sparingly.
- For chart morphing, we precompute the intermediate path data before the transition starts if possible, or use a single morph path rather than morphing dozens of individual bars (combining shapes reduces DOM nodes to animate).
- We also test on throttle settings to ensure that even mid-tier devices can handle the animations. If any animation consistently drops frames (e.g., complex SVG morph on mobile), we shorten its duration or simplify it to meet the budget.

- We include a “perf mode” switch (possibly automatically triggered by detecting low device performance or via user setting) that reduces effects (similar to reduced motion, but specifically if we detect e.g. low FPS in the first few animations, we can dynamically disable some animations).

- **Memory and Load Performance:** The architecture with RSC ensures initial bundle size is small (mostly just the code for interactive widgets, as static content is server-rendered). We use code-splitting: the 12 widget components are dynamically imported only if needed (especially if some sections are hidden or off-screen). The AI agent’s heavy libraries (if any, like a WebSocket client or large language model lib) are only loaded when the sidebar opens. We monitor bundle sizes and use Next.js analysis to keep each chunk lean. Also, we use webworkers for any computationally heavy tasks from the intelligence layer if needed (for example, if the AI does local computations or if a large data set needs sorting for a chart, a web worker can do it without blocking UI).

- **Telemetry Events:** We instrument the app with analytics and telemetry to gather usage data and detect issues:

- **User Behavior Events:** e.g., “WidgetReordered”, “WidgetResized”, “WidgetHidden”, “SidebarOpened”, “AIProposalApproved”. Each event includes metadata like user id (or anon id), timestamp, and relevant details (which widget moved, from where to where, etc.). These events help understand feature usage and can feed into improvements (e.g., if many users hide a particular default widget, maybe it’s not useful).
- **Performance Metrics:** We track client-side performance metrics such as Time to Interactive (from Next.js), animation frame rates (we might sample frame rate during drag and report if it dips below threshold), and memory usage if possible. We also capture offline usage metrics, e.g., how often users go offline, how many changes queued, average time to sync.

- **Error Logging:** If any error occurs (in syncing, data fetch, etc.), we log it (possibly via an error reporting service). For example, if an IndexedDB write fails or a sync returns a conflict, we record that event ("SyncError" with details).

- **Telemetry Pipeline:** All these events are batched and sent to our telemetry/analytics service. We ensure minimal impact on user experience by sending in background (navigator.sendBeacon or batching after render idle). We also respect user consent/config for telemetry.

- **Fallback Triggers:** We build in automatic fallbacks for extreme scenarios:

- If the app detects consistently poor performance (e.g., animations always dropping frames, or a device with low memory), we automatically enable the "reduced motion" and possibly simplify charts (maybe use static images or fewer data points) – essentially a **graceful degradation** mode. This could be triggered by a heuristic like "if drag FPS < 30 for 3 seconds, enable simple mode".
- If the AI service is unreachable (timeout or offline), the Sidebar agent shows a friendly message and possibly hides proposal functionality temporarily, rather than hanging. We also might trigger a retry or an alert icon on the AI toggle if it's down.
- If IndexedDB fails (some browsers in private mode might disable it), we fall back to in-memory storage for the session and warn the user that persistence is limited.
- In case our layout gets corrupted (e.g., an impossible overlap due to a bug), we have a "Reset layout to default" button that the user or system can trigger. Telemetry could detect if a layout load fails (e.g., JSON parse error or missing sections) and automatically reset to a known good state from the server.
- For data, if React Query cache is stale and network is down, rather than show outdated info without warning, we label data with a timestamp like "(as of 1h ago)" to inform the user. This is a UX fallback for data freshness.

10. Testing Plan

We adopt a comprehensive testing strategy covering unit tests, integration/UI tests, accessibility audits, and performance tests:

- **Unit Tests (Logic and Hooks):** We use Vitest (a Vite-friendly test runner) for unit testing our functions, hooks, and store logic:
- Test the Zustand store: initial state, state updates, and that actions (e.g., a function like `moveSection(id, newPos)` in the store) correctly update the state without mutating other sections.
- Test utility functions, e.g., grid collision resolver: feed in a layout and a proposed move, ensure it returns the expected new positions for affected widgets.
- Test hooks like `useOfflineSync`: simulate going online/offline by mocking `navigator.onLine` and ensure it flushes the queue appropriately, and that backoff timing logic works (we can advance timers in tests).
- Test the AI proposals logic: a function that applies an AIProposal to the state should produce the correct new layout and mark the proposal accepted.
- Test the context menu logic: ensure that given a right-click event at certain coords, the context menu opens with correct items, etc. (This might be covered more in integration tests, but basic logic like "calling onHide triggers the store update to hide that section" can be unit tested).

- Motion-specific units: if we have a function to generate an interpolated path for chart morph, test that given sample bar and line data, it returns a valid path string and preserves area, etc. (We might compare snapshots of path data).

- **Integration & E2E Tests (Playwright):** We write end-to-end scenarios using Playwright, which allows controlling the browser and making assertions on the rendered app. Key test scenarios:

- **Grid Operations:** Open the app with default 12-section layout. Simulate dragging one widget to a new position (Playwright can emulate mouse down, move, up). Verify that the widget's new position is saved (we can check that the DOM element moved or that the state in localStorage/IndexedDB updated via page.evaluate or by reloading the page and seeing the new order persists). Also verify no overlapping occurs: after drag, ensure no two widgets have the same grid coordinates.
- **Resize Widget:** Using the resize handle (if present) or a keyboard command if we allow that, simulate resizing a widget. Assert that the widget's DOM element grows/shrinks and that adjacent widgets reposition appropriately.
- **Sidebar Behavior:** Test toggling the Sidebar. Click the sidebar toggle button and assert that the sidebar appears (for overlay: it should overlay content; for push: main content's width/style should change). Then switch the mode: e.g., in settings choose "Push mode" and ensure that next open shifts the layout instead of overlaying. Also test that clicking outside the sidebar (or pressing Esc) closes it when in overlay.
- **AI Agent and Proposals:** Simulate typing a query to the AI agent (if the agent requires an API call, we might stub the network or use a mock response). Ensure the agent's response appears. Then simulate receiving a proposal (we might trigger an artificial proposal via test hook or by having the AI respond with a known suggestion). Click "Apply" on the proposal and verify the change took effect in the dashboard (e.g., a certain widget became visible or moved). Also ensure that after applying, the proposal entry is gone or marked accepted.
- **Context Menu:** Right-click (or keyboard shortcut) on a widget to open its context menu. Verify the menu is visible with correct options (e.g., "Hide Widget"). Click "Hide Widget" and confirm the widget is removed from the grid and a corresponding entry appears maybe in a "hidden widgets" list or that the layout has one less visible section. Also test that pressing Esc closes the context menu.
- We incorporate accessibility checks in E2E: e.g., after rendering the page, run axe-core via Playwright to catch any obvious issues (like missing ARIA labels or color contrast errors). We also simulate keyboard navigation: tab through elements and ensure focus goes in expected order (Playwright can track active element).

- **Offline/Sync E2E:** It's tricky but we can simulate offline in Playwright by toggling network conditions. Scenario: load the app, go offline (Page.setOfflineMode), perform some actions (move widget, etc.), then go online and verify that the changes eventually reflect on the server (maybe the server API in test records the update). Alternatively, check that the sync indicator shows pending offline and then clears after coming online.

- **Accessibility Testing:** Besides manual testing with screen readers (NVDA/VoiceOver) which we'll do in development, we automate some checks:

- Use **axe-core** in integration tests for each major view (dashboard loaded, sidebar open, context menu open) to detect color contrast issues, missing labels, etc.

- Use Testing Library's `jest-axe` in unit tests for isolated components. For example, render the Sidebar component and run `axe()` to ensure no a11y violations (with the caveat of ignoring known issues if any false positives).

- Test keyboard-only interaction: We can write a focused test that programmatically triggers keydown events. E.g., focus the first widget, press Tab and ensure focus moves logically through widgets, press Enter on a widget and ensure it maybe toggles something (if defined), press Esc to close things, etc. We verify expected element is focused at each step.

- **Performance & Motion Testing:** We set up a **motion test harness** where we reduce animation speeds for test determinism. For instance, we might configure Framer Motion with a reduced duration in test mode so we can simulate an animation quickly. We then:

- Test FLIP animation results: e.g., move a widget and during the animation, take snapshots of its position at start vs end. We can verify via DOM that it ends in the correct spot (the visual smoothness we trust FM for, but we ensure no final layout glitch).
- Chart morph test: perhaps unit test the path interpolation as mentioned. In a browser context, we could render a bar and line chart, toggle, and ensure after animation the SVG path matches the target shape.
- We use Lighthouse CI or Web Vitals in a testing environment to measure performance metrics (First Contentful Paint, etc.). This can be part of our CI pipeline to catch regressions in load performance.

11. Backlog (Jobs → Runs → Steps)

Below is an initial breakdown of the development tasks, structured as high-level **Jobs**, each broken into **Runs** (subtasks or sprint stories), and further into concrete **Steps**:

- **Job 1: Layout & Grid Foundation**
- *Run 1.1: Setup Grid Structure*
    - Step: Configure Tailwind grid (12 columns) and responsive breakpoints in CSS.
    - Step: Implement the `<Grid>` component with a 12-col CSS Grid and dummy sections for now.
    - Step: Verify basic responsive behavior (columns collapse on smaller screens).
- *Run 1.2: Section Container & Drag/Resize*
    - Step: Implement `<SectionContainer>` component that wraps content and can be positioned via inline styles (grid-area or absolute coords).
    - Step: Integrate a drag-and-drop library (e.g., `@dnd-kit`) or use Pointer events to make SectionContainer movable. Snap moves to grid coordinates.
    - Step: Implement resize handles on SectionContainer (perhaps at bottom-right corner) and allow resizing within grid constraints.
    - Step: Write collision detection logic to prevent overlaps and push other sections – ensure this logic triggers on drag/resize events.
    - Step: Test manually by dragging/resizing placeholders and observing layout adjustments.

• *Run 1.3: Persistence & State Management*

  - Step: Define Zustand `useDashboardStore` with state structure for sections layout (array of SectionConfig).
  - Step: Load initial layout (12 sections default) into the store on app startup.
  - Step: On any layout change (move/resize/hide), update the store and also persist to local storage/IndexedDB (implement a debounced save function).
  - Step: If a saved layout exists in storage, load it on init to restore user layout.
  - Step: Ensure that store updates trigger a re-render of the Grid with new positions (subscribe the Grid component to the store).

• **Job 2: Sidebar & AI Integration**

• *Run 2.1: Sidebar Component (Push & Overlay)*
  - Step: Implement `<Sidebar>` component UI (HTML structure with header, content list or chat area, footer).
  - Step: Add prop or context for `mode` and apply different Tailwind classes: in overlay mode, use `fixed left-0 top-0 h-full w-64` with a backdrop, in push mode use a static `<aside>` that occupies space next to content.
  - Step: Implement open/close behavior: perhaps controlled by Zustand or lifted state. Add a button in Header to toggle open state.
  - Step: Animate the open/close: use Framer Motion on the Sidebar container (slide in/out) and on the main content (for push mode translate).
  - Step: Test both modes manually and with different viewport sizes (simulate small vs large screen to possibly auto-switch to overlay on mobile).
• *Run 2.2: AI Agent Component*
  - Step: Create `<SidebarAgent>` component inside Sidebar. Include a text input for queries and an output area for answers/proposals.
  - Step: Integrate a call to the backend AI API: on form submit, send query (using React Query mutation or plain fetch). Use a loading state to indicate thinking, then display answer text.
  - Step: Display AI proposals if any in a list. For now, simulate proposals or use a stubbed response structure.
  - Step: Implement Approve/Reject buttons for proposals that call provided callbacks. The approve callback should dispatch an action to apply changes (e.g., call store to update layout) and remove the proposal from list.
  - Step: Ensure the SidebarAgent is only visible when Sidebar is open and maybe hidden or unmounted when closed (to avoid running queries when not needed).

• *Run 2.3: Sidebar Keyboard & Focus*

  - Step: Add focus trap logic for overlay mode (maybe use `focus-trap-react` library or custom). Write logic so that opening sidebar focuses the input field, and closing returns focus to origin.
  - Step: Add `aria-*` attributes to Sidebar (role, aria-modal for overlay, labelledby for header).
  - Step: Add keyboard shortcut (e.g., Ctrl+K or just a designated key) to open Sidebar for convenience.
  - Step: Test keyboard navigation: Tab should cycle within sidebar when open (for overlay), Esc closes it.

• **Job 3: Widget Functionality & Context Menus**

• *Run 3.1: Widget Content & Types*
  ◦ Step: Create placeholder components for different widget types (ChartWidget, TableWidget, etc.), each taking some sample data and rendering via visx/d3.
  ◦ Step: Integrate real data fetching with React Query: for each widget type, create a query hook (e.g., `usePortfolioData` for a portfolio chart) that fetches from an API route. Use SSR data if available (Next.js can provide via server components).
  ◦ Step: Render each SectionContainer's children by mapping section type to the corresponding Widget component, passing necessary data.
  ◦ Step: Ensure that if data is loading or error, show a spinner or message inside the widget container.
• *Run 3.2: Context Menu Implementation*
  ◦ Step: Implement a generic `<ContextMenu>` component that uses Portal to render a menu overlay. It accepts items and coordinates.
  ◦ Step: In SectionContainer, add an invisible trigger (or right-click handler) to open the ContextMenu for that section. On right-click or on a "menu" button click, populate menu items like "Refresh Data", "Hide Widget", "Settings".
  ◦ Step: Hook up menu item actions: "Hide Widget" sets section.visible=false via store, "Refresh" could refetch the query (React Query's `invalidateQueries` for that widget), "Settings" might open a stub modal.
  ◦ Step: Ensure the context menu closes on selection or on clicking outside. Also support keyboard opening: if Section has focus, pressing Shift+F10 or the context menu key opens it.

• *Run 3.3: Drag, Drop & Resize Polish*

  ◦ Step: Refine drag behavior: constrain within grid bounds, maybe highlight potential drop cells. Use dnd-kit collision detection to swap or push items in real-time as you drag.
  ◦ Step: Refine resize: enforce a min and max size for each widget type (e.g., a chart might have min width 3 cols, etc.), and live-update the layout as resizing.
  ◦ Step: Add visual guides (like a ghost outline of new size/position while dragging/resizing).
  ◦ Step: Write unit tests for the collision logic to ensure no overlaps and correct shifting.

• **Job 4: Persistence, Offline & Sync**

• *Run 4.1: IndexedDB Persistence*
  ◦ Step: Set up `idb` library and create an `IndexedDBDatabase` instance with object stores: e.g., `layoutDraft`, `changeQueue`.
  ◦ Step: Implement functions saveLayoutToIDB and loadLayoutFromIDB in a persistence utility module. Use those in the autosave flow (from Job 1.3).
  ◦ Step: Also use `persistQueryClient` from React Query: set up a persister (using `react-query-persist-client` with IndexedDB). Confirm that query data is written (can test by looking into IDB after data loads).
  ◦ Step: Test by turning off network and refreshing page: cached data should load from IDB, and layout from IDB as well.

• *Run 4.2: Offline Change Queue*
  ◦ Step: Implement a Zustand store or context to manage the outbox queue (or simply use the IDB directly with events). Possibly use React Query's offline mutation persistence as a starting point.
  ◦ Step: Write a function `enqueueChange(change)` that writes a change entry to IDB. This is called whenever a layout change occurs while offline (or always, marking it synced if online).
  ◦ Step: Write a sync processor `flushQueue` that reads the queue and attempts to POST/PUT changes to server. Use a recursion or loop with await to preserve order.
  ◦ Step: Listen to `window.online` event and call `flushQueue`. Also, in React Query's persist config, utilize `onSuccess` to resume mutations [6].
  ◦ Step: Implement retry with backoff: use setTimeout with incremental delays if a sync fails. Possibly integrate with React Query's `resumePausedMutations` as well.

• *Run 4.3: Sync Indicator & Conflict Handling*

  ◦ Step: Add UI element in Header (e.g., an icon) that reflects `navigator.onLine` status. Style it with a red outline if offline.
  ◦ Step: If `useDashboardStore` has unsynced changes count, display that number on the icon as a badge.
  ◦ Step: Write logic to update this count: increment when enqueueing a change, decrement when a change is successfully synced and removed from queue.
  ◦ Step: Handle basic conflict: if server responds with a conflict or out-of-sync error, decide on strategy (for now, assume last write wins – we overwrite server with client state, or fetch server layout and merge if needed). In conflict, log it and perhaps reset local layout to server's version for safety, notifying user.

• **Job 5: Quality Assurance**

• *Run 5.1: Accessibility Review*
  ◦ Step: Go through each UI element and add missing ARIA labels/roles. Create an `a11y.ts` config if needed mapping keys (like keyboard shortcuts usage via `aria-keyshortcuts`).
  ◦ Step: Test with screen reader (NVDA/VoiceOver) navigating the dashboard: ensure it announces section titles, can operate context menu, etc. Fix any issues (like adding `aria-haspopup="menu"` on context menu buttons, etc.).
  ◦ Step: Implement any needed focus management adjustments discovered (like ensuring modals focus trap).
  ◦ Step: Run automated axe and fix issues (color contrast, missing alt attributes, etc.).
• *Run 5.2: Performance Tuning*
  ◦ Step: Run Lighthouse on the app in production mode. Check metrics for performance. Optimize any slow points (e.g., large bundles – maybe apply code splitting or tree shaking if needed).
  ◦ Step: Profile drag and drop with DevTools performance monitor to ensure 60fps; optimize by removing any causing reflows (e.g., if updating state too often, throttle it).
  ◦ Step: Test on a low-end device or simulator to ensure acceptable experience.
• *Run 5.3: Automated Test Suite*
  ◦ Step: Write unit tests for critical modules (store, utils) as outlined above.
  ◦ Step: Configure Playwright and write E2E tests for major user flows (drag-drop, sidebar, offline scenario). These might involve setting up a test server API or mocking responses.

- Step: Integrate tests with CI pipeline (GitHub Actions or similar) to run on each commit. Ensure that any failing a11y checks or unit tests block the build.
- Step: Additionally, add a visual regression test for the grid layout (e.g., use Playwright's screenshot comparison to catch any layout breaking changes in the 12-section grid).

This backlog serves as a starting WBS (Work Breakdown Structure) for implementation tasks, which can be adjusted and expanded as needed during development.

12. Scaffold Code

Below is a minimal scaffolding of the project's front-end structure with placeholder files and stub implementations, following the Next.js App Router conventions and the specified stack:

```
app/
├─ layout.tsx            // Root layout defining Header, Sidebar, Footer
wrappers
├─ page.tsx              // Dashboard page component (could fetch initial data)
├─ dashboard/            // (Optional) route group for dashboard-specific files
│   └─ grid.tsx          // (Optional: could also be in components)
public/
├─ ... static assets ...
components/
├─ Header.tsx            // Header component
├─ Footer.tsx            // Footer component
├─ Sidebar/
│   ├─ Sidebar.tsx       // Sidebar container component
│   ├─ SidebarAgent.tsx  // AI agent UI inside the sidebar
│   └─ Sidebar.module.css// Sidebar-specific styles (if any, or use Tailwind)
├─ Grid.tsx              // Grid component that lays out SectionContainers
├─ SectionContainer.tsx  // Container for each dashboard section (draggable/
resizable)
├─ widgets/
│   ├─ ChartWidget.tsx   // Example widget content component (uses visx for
charts)
│   ├─ TableWidget.tsx   // Example widget (table of data)
│   └─ ...other widgets...
├─ ContextMenu.tsx       // Context menu component
└─ IconButton.tsx        // Reusable small button (could be used for context
menu trigger, etc.)
hooks/
├─ useDashboardStore.ts  // Zustand store for dashboard layout and state
├─ useOfflineSync.ts     // Hook to manage offline queue sync (listens to
connectivity)
├─ useKeyboardShortcuts.ts // Hook to register global keyboard shortcuts (e.g.,
for sidebar toggle)
└─ useReducedMotion.ts   // Hook to check prefers-reduced-motion and provide
```

```
boolean
utils/
├─ gridUtils.ts          // Utility functions for grid math (collision
detection, etc.)
├─ motionUtils.ts        // Helper for FLIP animations or framer motion variants
├─ idbUtils.ts           // IndexedDB helper functions (save layout, load
layout, queue ops)
└─ aiUtils.ts            // Perhaps formatting or handling AI proposal data
styles/
└─ globals.css           // Tailwind base imports and any global styles
tests/
├─ unit/
│   ├─ dashboardStore.test.tsx
│   ├─ gridUtils.test.ts
│   └─ useOfflineSync.test.ts
├─ e2e/
│   ├─ dashboard.e2e.ts   // Playwright tests for dashboard interactions
│   └─ offline.e2e.ts     // Playwright tests for offline scenario
└─ a11y/
    └─ axe.test.tsx       // Automated accessibility tests (using axe)
```

Example Stub Implementations (TypeScript):

File: `components/Sidebar/Sidebar.tsx` – Sidebar container with push/overlay handling:

```tsx
"use client";
import { FC, useEffect } from 'react';
import { SidebarProps } from '../../components.types';  // assume we export
interfaces from a types file or index

export const Sidebar: FC<SidebarProps> = ({ mode, open, onToggle, width = 300,
children }) => {
  // Apply different classes based on mode
  const baseClasses = "h-full bg-gray-800 text-white transition-all";
  const overlayClasses = "fixed top-0 left-0 z-50 shadow-xl " + (open ?
"translate-x-0" : "-translate-x-full");
  const pushClasses = "absolute top-0 left-0 " + (open ? "translate-x-0" : "-
translate-x-full");
  // ^ In push mode, Sidebar could actually be static in layout (not absolute),
but for animation we treat it similarly.

  // Lock body scroll when overlay sidebar is open
  useEffect(() => {
    if (mode === 'overlay') {
      document.body.style.overflow = open ? 'hidden' : '';
    }
```

```
  }, [mode, open]);

  return (
    <aside
      className={
        baseClasses + " " + (mode === 'overlay' ? overlayClasses : pushClasses)
      }
      style={{ width: typeof width === 'number' ? `${width}px` : width }}
      aria-hidden={!open}
    >
      {/* Sidebar content (header, agent, etc.) */}
      <div className="flex flex-col h-full">
        {/* Header with close button */}
        <div className="p-4 font-bold flex justify-between items-center">
          <span>AI Assistant</span>
          <button onClick={onToggle} aria-label="Close Sidebar">X</button>
        </div>
        {/* Sidebar body (AI agent) */}
        <div className="flex-1 overflow-y-auto">{children}</div>
      </div>
    </aside>
  );
};
```

File: `components/SectionContainer.tsx` – Draggable, resizable section wrapper:

```
"use client";
import { FC, useRef } from 'react';
import { SectionContainerProps } from '../components.types';
import { useDashboardStore } from '../hooks/useDashboardStore';

export const SectionContainer: FC<SectionContainerProps> = ({ config, children,
onDragEnd, onResize, onHide, onOpenContextMenu }) => {
  const { id, position, size, title, visible } = config;
  const ref = useRef<HTMLDivElement>(null);

  // Derive CSS grid position if using CSS Grid positioning:
  const style: React.CSSProperties = {
    gridColumnStart: position.x + 1,              // assuming position.x is 0-
indexed
    gridColumnEnd: `span ${size.w}`,             // span w columns
    gridRowStart: position.y + 1,
    gridRowEnd: `span ${size.h}`
  };

  if (!visible) {
```

```
    return null; // hidden sections are not rendered
  }

  return (
    <div
      ref={ref}
      className="section shadow-sm border border-gray-300 bg-white rounded-md
relative"
      style={style}
      data-section-id={id}
      tabIndex={0}
      aria-label={title}
      onContextMenu={e => {
        e.preventDefault();
        onOpenContextMenu && onOpenContextMenu(id, { x: e.clientX, y:
e.clientY });
      }}
    >
      {/* Content */}
      <div className="section-content p-2">{children}</div>
      {/* Resize handle (bottom-right corner) */}
      <div
        className="resize-handle"
        onMouseDown={() => { /* initiate resize logic, perhaps using a hook or
direct events */ }}
      />
      {/* Context menu trigger button (for keyboard users) */}
      <button
        className="absolute top-1 right-1 p-1 text-gray-500"
        onClick={e => onOpenContextMenu && onOpenContextMenu(id, { x:
e.clientX, y: e.clientY })}
        aria-label={`Open menu for ${title}`}
      >⋮</button>
    </div>
  );
};
```

File: `hooks/useDashboardStore.ts` – Zustand store for layout and actions:

```
import { create } from 'zustand';
import { SectionConfig } from '../components.types';
import { persistLayoutToIDB, enqueueChange } from '../utils/idbUtils';

interface DashboardState {
  sections: SectionConfig[];
  // Actions to modify state
```

```
  moveSection: (id: string, newPos: {x: number, y: number}) => void;
  resizeSection: (id: string, newSize: {w: number, h: number}) => void;
  hideSection: (id: string) => void;
  setSections: (sections: SectionConfig[]) => void;  // load initial or after
sync
}

export const useDashboardStore = create<DashboardState>((set, get) => ({
  sections: [],  // initial empty, will be set after load
  moveSection: (id, newPos) => {
    set(state => {
      const sections = state.sections.map(sec =>
        sec.id === id ? { ...sec, position: newPos } : sec
      );
      return { sections };
    });
    const updatedSec = get().sections.find(sec => sec.id === id);
    // Persist and enqueue change (if offline)
    persistLayoutToIDB(get().sections);
    enqueueChange({ type: 'move', id, newPos });
  },
  resizeSection: (id, newSize) => {
    set(state => {
      const sections = state.sections.map(sec =>
        sec.id === id ? { ...sec, size: newSize } : sec
      );
      return { sections };
    });
    persistLayoutToIDB(get().sections);
    enqueueChange({ type: 'resize', id, newSize });
  },
  hideSection: (id) => {
    set(state => {
      const sections = state.sections.map(sec =>
        sec.id === id ? { ...sec, visible: false } : sec
      );
      return { sections };
    });
    persistLayoutToIDB(get().sections);
    enqueueChange({ type: 'hide', id });
  },
  setSections: (sections) => set({ sections })
}));
```

File: `utils/gridUtils.ts` – (simple example collision check utility):

```
import { SectionConfig } from '../components.types';

/** Checks if a section occupies the given grid cell */
function occupies(section: SectionConfig, x: number, y: number): boolean {
  return section.position.x <= x && x < section.position.x + section.size.w &&
         section.position.y <= y && y < section.position.y + section.size.h;
}

/** Find the next free position (scanning downwards) for a section of given size
*/
export function findNextFreePosition(sections: SectionConfig[], desired:
{x:number,y:number}, size: {w:number,h:number}): {x:number,y:number} {
  let { x, y } = desired;
  // Loop until found space
  // (For simplicity, increment row by row)
  while(true) {
    // Check all cells of the proposed area
    let collision = sections.find(sec => sec.visible &&
      // any overlap between sec and the area [x,x+w) x [y,y+h)
      !(sec.position.x >= x+size.w || sec.position.x+sec.size.w <= x ||
        sec.position.y >= y+size.h || sec.position.y+sec.size.h <= y)
    );
    if (!collision) {
      return { x, y };
    }
    // If collision, move down past the collision's bottom
    y = collision.position.y + collision.size.h;
    // Optionally, reset x to 0 if we always fill from left (depends on
strategy)
    x = desired.x;
    // (This is a naive algorithm and can be improved)
  }
}
```

This scaffold code outlines the basic structure and some logic in a simplified form. In a real implementation, we would flesh out these stubs with full functionality and error handling, but this provides a blueprint of how components, state, and utils are organized.

13. Acceptance Criteria

To declare the module “done”, the following key behaviors must be validated (each mapping to requirements like layout, reduced motion, offline, etc.):

- **Dashboard Layout & Widgets:** The dashboard displays 12 sections by default, arranged in a responsive grid. **Acceptance Test:** On a desktop viewport, exactly 12 section widgets are visible (unless some were intentionally hidden), arranged without overlap. The user can drag a widget to a

new position; upon dropping, the widget smoothly animates to its new slot and no two widgets overlap (others shift out of the way if needed). After a page reload, the widget remains in its new position (layout persisted). If the user resizes a widget (by dragging its corner), the widget's content reflows and neighboring widgets adjust position immediately. On a mobile viewport, sections stack in one column and remain in user-defined order.

- **Reduced Motion Mode:** The application honors the user's reduced motion preference. **Acceptance Test:** In a browser with `prefers-reduced-motion` enabled, opening the sidebar or reordering widgets happens instantly with no sliding or morphing animation (or uses minimal fade if necessary). In standard mode, these actions are animated (sidebar slides in/out, charts morph smoothly). We can programmatically toggle the preference and verify that in reduced motion mode, the CSS or JS skips animations (e.g., Framer Motion's animations complete immediately or a CSS class like `.reduce-motion` is applied to disable transitions).

- **Offline Persistence & Sync:** The module supports offline use and syncing changes. **Acceptance Test:** Use the app offline (simulate by disabling network): The dashboard still shows the last loaded data in each section (no errors). Perform several changes: move a widget, hide another, and approve an AI proposal. The UI should reflect these changes immediately. A "offline changes pending" indicator becomes visible (e.g., showing 3 pending changes). Close and reopen the app while still offline – the changes persist (thanks to IndexedDB draft). Then restore network connectivity: the pending changes are sent to the server (simulate or check via logs), and the indicator clears automatically. After a refresh (with network on), the server's state matches all the changes made offline. No data is lost, and unsynced changes do not revert.

- **AI Proposal Approval Flow:** AI-generated suggestions require user approval and correctly apply on acceptance. **Acceptance Test:** Trigger a situation where the AI agent produces a proposal (for example, it suggests "Hide section X"). The proposal appears in the sidebar with a clear description. Before approval, no change is made to the dashboard (section X is still visible). Upon clicking "Approve" (or via keyboard, selecting the proposal and pressing Enter), the proposal is marked accepted/removed, and the suggested changes are enacted: in this case, section X becomes hidden in the layout. The layout state updates and persists just as if the user performed the hide action themselves. If the user clicks "Reject", the proposal is dismissed and no changes occur to the layout. The system should also prevent conflicting proposals from applying simultaneously (if two proposals try to move the same widget, only the one accepted by the user actually changes state). This criteria is met when an approved proposal reliably changes the UI state and a rejected one does nothing, with the appropriate update to the proposals list.

- **General Acceptance (Regression Checks):** Additionally, the app should meet general quality bars: All interactive elements can be reached and operated via keyboard alone; important UI actions (sidebar toggle, context menu, etc.) have no console errors during use; and the initial load of the dashboard is under a reasonable time (e.g., <2s for first content, which will be validated via performance tests not directly by a user story, but is implicitly required).

Each of these criteria will be verified during final testing/demo to ensure the module is feature-complete and behaves as expected under various conditions.

14. Risks & Mitigations

During development, we identified potential risks and have planned mitigations for each:

- **Risk: Chart Morph Jank:** The bar-to-line chart morph animation could be visually jarring or drop frames, especially with complex SVG paths or on slower devices. **Mitigation:** We will simplify the morph by using a single path interpolation instead of morphing many individual elements. We leverage GPU-friendly transforms where possible. If performance is still an issue, we implement a fallback of a cross-fade between a static bar chart image and line chart image for that transition on low-end devices or when `prefers-reduced-motion` is on. We will test the morph with large data sets and on mobile to ensure it remains smooth, adjusting duration or easing to hide minor jank.

- **Risk: Offline Data Corruption:** Data saved in IndexedDB (layout or queued changes) could become corrupt or out-of-sync (for example, if the schema changes between app versions, or a user uses the app in two tabs leading to conflicts). **Mitigation:** We version our IndexedDB schema and provide migration functions if we ever change the structure. We also guard all IDB operations with try/catch and, on error, we can fall back to a safe default (for instance, if loading layout fails, we log the error and reset to default layout, possibly informing the user). To handle multi-tab scenarios, we could use the Broadcast Channel API or Zustand's `storage.onDidChange` if available to sync state across tabs, ensuring only one source of truth. Our change-set ledger can also be used to reconcile differences by replaying missing changes. In worst-case scenarios (corruption), we detect it and prompt the user to refresh or reset (and perhaps upload error telemetry so we can fix the root cause).

- **Risk: Keyboard Shortcut Conflicts:** Our chosen keyboard shortcuts might interfere with browser defaults (like Ctrl+K is often "search" in web apps, or single letters can conflict with screen reader commands). **Mitigation:** We carefully choose uncommon combos (e.g., Ctrl+Shift+A for opening the AI sidebar, instead of something common). We also allow shortcuts to be remapped in settings if this is a concern for users. During testing, we verify no key presses inadvertently trigger browser behaviors (for example, if we used `?` for help, ensure it doesn't input a character into an input field). All shortcuts are documented so users know what's available, and we provide the ability to disable them if needed (some users may not want any global hotkeys from the app interfering with other software).

- **Risk: Layout Thrashing on Drag:** If our drag implementation updates state on every mouse move, it could cause continuous re-renders and layout thrash, slowing the drag or freezing the UI. **Mitigation:** We will throttle drag updates (e.g., update state at most 60fps using `requestAnimationFrame` or even every 2-3 frames if needed). Framer Motion's layout animations inherently handle this by animating position without constant reflows. If using dnd-kit, we'll use its built-in sensors for collision rather than manually forcing layout calc each pixel of movement. We'll also test large layouts to ensure performance holds up, adjusting strategy (like hiding a dragging element from flow and only dropping it at the end to reduce re-layout calculations during drag).

- **Risk: Excessive Memory Usage:** Storing 12 widgets worth of data (especially if they include charts with large datasets) and keeping multiple versions (due to the change log) could grow memory and storage usage, possibly causing performance issues or hitting browser storage limits. **Mitigation:**

We will paginate or limit data stored in the front-end for each widget (e.g., charts might only request a summary or last N points rather than entire history, as appropriate). The React Query cache will be configured with reasonable max age and size limits. The change-set ledger in IDB will be pruned – for example, after a successful sync, we might keep only the last 20 changes or those from the last week for audit, deleting older ones. Also, using IndexedDB for caching avoids the 5MB localStorage limit and is more efficient; still, we will monitor storage size and if it grows beyond, say, 50MB, consider clearing old data (with user confirmation if needed). Telemetry events will be batched and not stored long on client, so they should not accumulate too much.

- **Risk: Unsynced Changes Edge Case:** If a user makes offline changes and then a separate change on another device before the first device comes online, conflicts may occur (last write wins could override something unexpectedly). **Mitigation:** While full conflict resolution is complex, our approach ensures that when coming online, the client fetches the latest server state before applying queued changes – if it detects a fundamental mismatch (like section count differs), it can alert the user or merge intelligently (perhaps by prioritizing user's offline changes but still pulling in new sections added elsewhere). Since this app is primarily single-user focused, this scenario might be rare, but we handle it by not blindly overwriting server state: each queued change could be reapplied on top of a fresh server-fetched layout state, then send that final state back to server. Additionally, we include a timestamp with changes; the server can use that to decide to accept or flag conflicts.

- **Risk: Third-Party Integration Failures:** Dependencies like the AI API or visx charts might fail or have breaking changes. For example, the AI service could timeout or return unexpected results. **Mitigation:** We encapsulate calls to external services with timeouts and error handling. If the AI call fails, we catch it and display a graceful error in the sidebar ("The assistant is taking a break, please try again later") without crashing the app. For visx/d3, if a rendering error occurs (say, data undefined), we catch in the component and perhaps show a placeholder message instead of a blank screen. We will lock dependency versions and test after upgrades.

By anticipating these and other risks, and implementing the mitigations above, we aim to deliver a robust, user-friendly dashboard front-end that handles edge cases gracefully and maintains a high standard of quality.

---

1 Next.js 15 App Router — Architecture and Sequence Flow | by Suresh Kumar Ariya Gowder | Medium
https://medium.com/@sureshdotariya/next-js-15-app-router-architecture-and-sequence-flow-3a6ffdd2445c

2 3 5 6 Powering offline-ready web apps with React Query
https://www.kylereblora.com/posts/powering-offline-ready-apps-with-react-query

4 How the Washington Post design system made me learn about Perceptual Contrast (APCA) - Steve Frenzel
https://www.stevefrenzel.dev/posts/learning-about-perceptual-contrast/