ChatGPT

Cyber Trust Sensor Dashboard – Architecture Analysis

Current Application Structure and Functionality

Overview: The Cyber Trust Sensor Dashboard is a React application that consolidates multiple cybersecurity-related tools and views into a single dashboard interface. The core of the app is the **RequirementsDashboard** React component, which acts as the main container for the entire dashboard UI and logic. This component is responsible for rendering various sections (requirements management, capabilities, threat intelligence, risk management, standards compliance, etc.) based on the current view state. It integrates data from multiple sources (currently mocked or static) and provides interactive features such as filtering, searching, and editing of requirements and capabilities.

Main Components and Layout: The dashboard's UI is composed of several React components organized by feature:

- **RequirementsDashboard.jsx:** A large component (~3500 lines) that holds the primary layout and state. It uses a reducer (`useReducer`) to manage a centralized UI state for the dashboard (e.g., current view mode, filters, modals) [1] [2]. It conditionally renders different sections of the dashboard based on a `viewMode` state (such as *overview*, *capabilities*, *requirements*, *analytics*, *PCD breakdown*, *maturity analysis*, *threat intelligence*, *MITRE ATT&CK navigator*, *risk management*, *standards*, *settings*, etc.). Each section may have its own sub-components. For example, when `viewMode === 'requirements'`, the **RequirementsTable** component is rendered to display and manage the list of requirements [3].

- **RequirementsTable.jsx:** A component that displays a paginated table of requirements and provides UI controls for filtering, searching, importing/exporting CSV, etc. [4] [5]. It receives from **RequirementsDashboard** the list of all requirements and the list of currently filtered requirements, as well as callbacks for events like filtering or viewing a specific requirement. This separation allows the table to focus on presentation logic (rendering rows, handling pagination and local UI state for toggling filter panels) while deferring data management (e.g., applying filters) to the parent or to hooks.

- **Capability List (in RequirementsDashboard):** The dashboard's **Capabilities View** shows a grid of capability cards, each summarizing a security capability and its associated requirements. Each capability card displays the capability's name, description, status, completion rate of its requirements, and key metrics like business value and ROI [6] [7]. Clicking a capability triggers a handler that updates the state to focus on that capability's requirements (by setting the selected capability filter and switching to the requirements view) [8]. This effectively drills down into the requirements specific to that capability.

- **Modals and Subsystems:** The app uses modal dialogs for editing or viewing details. For instance, **EditRequirementModal**, **ViewRequirementModal**, **NewCapabilityModal**, **CSVUploadModal**, etc., are imported and conditionally rendered by RequirementsDashboard when needed. These modals allow users to add or modify requirements and capabilities. The

state's `ui` slice contains flags like `showNewCapabilityModal`, `showUploadModal`, etc., to control the visibility of these modals [9] [10].

- **Other Feature Components:** There are dedicated components for specialized views:

- **PCDBreakdownView:** Likely presents a breakdown of People, Process, and Technology (or similar "PCD") data across capabilities.
- **BusinessValueView:** Visualizes requirements by their business value or justifications.
- **MaturityAnalysisView:** Analyzes requirements or capabilities in terms of maturity levels.
- **ThreatIntelligenceSystem & MitreAttackNavigator:** Provide threat intelligence data and a MITRE ATT&CK matrix view for threat coverage.
- **RiskManagement.jsx:** Manages operational and security risk items, including risk creation, updates, comments, and an audit trail [11] [12]. It maintains its own internal state via a reducer (separate from the main dashboard state) to handle a list of risk entries [13].
- **SystemSettings & DiagnosticsView:** Provide configuration settings and system diagnostics information (as implied by the inclusion of a *settings* view and diagnostic data).
- **StatCard, MaturityIndicator, InteractiveChart, LoadingSpinner, Portal/Modal infrastructure:** Reusable UI components for displaying statistics, loading states, interactive charts (using `recharts` library), and handling modals (Portal component for modals).

Data Flow: The application currently uses **custom React hooks** to handle data retrieval and state for different domains: - **useRequirementsData:** Provides the list of requirements and functions to add, update, delete, or import requirements [14]. Under the hood, this likely uses in-memory data or local storage to persist requirements (since no external API calls are evident in the provided code). Initially, it may generate mock requirement data (via utility functions like `generateMockData`) to populate the dashboard. - **useCapabilitiesData:** Similarly provides the list of capabilities and a function to add a new capability [15]. Each capability has an ID, name, status, description, and possibly links to requirements (via a `capabilityId` field on requirements). - **usePCDData:** Supplies data for "PCD" (possibly stands for *Policies/Controls/Devices* or some framework) along with update functions [16]. - **useAnalytics:** Processes the requirements list to derive summary analytics (for example, count of requirements by status, distribution of business value scores, etc.) [17]. This data is used to render charts in the Analytics view (e.g., a bar chart of requirement status counts and a scatter chart for business value vs. cost) [18] [19]. - **useFilteredRequirements:** Applies the current filter criteria and search term to the full requirements list to produce the `filteredRequirements` subset [20]. This encapsulates the filtering logic outside of the component, making the component code cleaner when determining which requirements to display.

All these hooks abstract data handling such that **RequirementsDashboard** doesn't need to know if data comes from an API, localStorage, or hardcoded mocks. In the current state, these hooks likely use static or mock data sources (e.g. generating mock capabilities and requirements via utilities).

Global Contexts and Providers: The application uses React Context for cross-cutting concerns: - **ThemeContext (ThemeProvider):** Provides theming information. The dashboard obtains the current theme (e.g. a "stripe" theme vs default) via `useTheme()`, and adjusts CSS classes accordingly [21] [22]. For example, theme context supplies `themeClasses` (like `mainContainer`, `sidebar`, `header` styles) so that switching themes can globally restyle the layout without changing each component [23]. - **AuthContext (JWTAuthProvider):** Likely provides authentication and current user info. **RequirementsDashboard** imports `AuthContext` [24] and possibly uses it to get the current user's details or to conditionally show data (e.g., to stamp created/updated by user, or to restrict certain actions based on role). In the RiskManagement component, for instance, `currentUser` is used to tag comments and audit trail entries [25] [26], which implies `currentUser` comes from an AuthContext

accessible within those components. - **Toast Context (useToast):** Provides `addToast` for showing user notifications [27]. Throughout the dashboard, this is used to display success or error messages (e.g., after importing or exporting data, a toast is shown confirming the action [28] [29]). - **Portal Context (usePortal):** Used for rendering modals in a React Portal (outside the normal component hierarchy). The `Portal` and `Modal` components [30] likely rely on a portal context to place modals at the end of the DOM for accessibility. This is part of the **Portal System** which enables modals to render on top of the UI.

State Management: The dashboard manages state at multiple levels: - **Local Component State:** Components like RequirementsTable use local `useState` for UI toggles (e.g., showing filter panel or column selector, current page number in pagination) [31] [4]. This state affects only the component's presentation and is not needed globally. - **Dashboard UI State (useReducer):** RequirementsDashboard defines an `initialState` object and a `dashboardReducer` to handle complex UI state across the dashboard [32] [2]. This state includes: - Filters for requirements (area, type, status, priority, maturityLevel, applicability, capability filter) [32]. - A nested `ui` object for view mode and UI controls (e.g., `viewMode` for current section view, `sidebarExpanded` toggle, `chartFullscreen` id, and flags to show/hide various modals and panels like `showUploadModal`, `showNewCapabilityModal`, `showProfileSetup`, etc.) [33] [9]. - A `modal` object to manage the currently open requirement modal (which requirement is selected for viewing/editing) [34]. - Search term and column visibility settings for the requirements table [35]. - A `standards` object to manage the state of the Standards & Frameworks section (e.g., for NIST CSF, it holds assessment data and which function/category is expanded or selected) [36].

The reducer handles dozens of action types to update this state in an immutable way. For example, `SET_FILTER` updates a filter value [37], `TOGGLE_NEW_CAPABILITY_MODAL` flips the flag to show/hide the New Capability modal [38], `SET_VIEW_MODE` switches the active view page [39], and `SET_SELECTED_CAPABILITY` not only stores the chosen capability ID but also sets the filter to that capability (so that the Requirements view will be scoped) [40]. There are also actions to handle the standards assessment data (e.g., updating NIST CSF assessment results) [41] [42].

- **Shared State via Hooks:** The data-providing hooks (useRequirementsData, useCapabilitiesData, etc.) internally manage state (likely with their own `useState` or `useReducer`). For instance, useRequirementsData might maintain the list of requirement objects and expose functions to mutate them. These hooks encapsulate that state and provide accessors/updaters. The RequirementsDashboard subscribes to these via the hook return values. In effect, this is a form of store – for example, the list of requirements acts as a data store that multiple components (table, charts, etc.) consume. However, currently these hooks appear to be **local to the component and not global singletons** (meaning if two different components call useRequirementsData, they might get their own data unless the hook itself uses a Context or singleton module).

Constants and Utilities: The app has a set of constants and utility modules that define static reference data and helper functions: - **Constants:** In a `constants` directory, static configuration objects are defined. For example, `companyProfile.js` under constants might list categories like `COMPANY_SIZE_THRESHOLDS`, `ANNUAL_REVENUE_RANGES`, `EMPLOYEE_COUNT_RANGES`, and compliance frameworks lists [43]. Similarly, threat and risk constants (threat levels, risk status values, etc.) are defined in a constants or utils file [44] [45]. These constants centralize fixed values used across components (e.g., mapping a status to a color or a label). In the RequirementsTable, a constant `PROGRESS_STATUSES` is imported and used to style progress status pills [46] [47]. This approach improves maintainability by avoiding magic strings or duplicated values in components. - **Utility Functions:** The `utils` directory contains domain-specific logic abstracted into functions: -

companyProfile utils: e.g., `determineCompanySize` computes a company's size category based on thresholds [48], `getSuggestedFrameworks` picks recommended compliance frameworks for a given company size [49], and various validators for profile data are provided [50]. - **CSV utilities:** Functions like `parseCSV`, `generateCSV`, `downloadCSV` handle converting requirement data to/from CSV and triggering downloads [51]. - **Data generation services:** `dataService.js` provides `generateMockCapabilities` and `generateMockData` which create sample capabilities and requirements data for initial load [52]. These are used when an external API or database is not yet connected, allowing the app to run with realistic dummy data. - **Risk/Threat utils:** e.g., `calculateRiskScore`, `getThreatLevel` in **threatProcessing.js** compute risk metrics [53] [54], and mapping functions might convert internal data to standard formats (e.g., a MITRE ATT&CK mapping to STIX format) [55] [56].

All such utilities and constants promote reusability and clarity by separating pure logic from component implementation. Components import what they need (e.g., risk calculation or profile helpers) to display derived values or perform validations.

Relationships Between Architecture Layers

The current codebase can be conceptually divided into the following layers or categories, each with distinct responsibilities and interactions:

- **Components (UI Layer):** These are the React components responsible for rendering the user interface. Components can be further sub-categorized:
- **UI Elements (Presentational Components):** e.g., Buttons, icons (from Lucide React), small components like **StatCard**, **MaturityIndicator**, etc., which correspond to single pieces of UI. They receive props and display information (possibly in an atomic design sense, these would be *atoms* or *molecules*).
- **Composite Components (Containers/Organisms):** e.g., **RequirementsTable**, **RiskManagement**, **ThreatIntelligenceSystem**. These bring together multiple UI elements and embed logic specific to a feature. For instance, RequirementsTable contains table layout, filter inputs, and uses helper callbacks for events but defers actual data changes to higher-level handlers.
- **Page/Root Component: RequirementsDashboard** itself, which assembles the entire page (like an *organism* or even *template* in atomic design terms). It includes the sidebar navigation, header, and whatever view is active in the content area. It manages high-level state and passes relevant slices down as props to child components.
- **Modal Components:** e.g., **EditRequirementModal**, **NewCapabilityModal** – though implemented as React components, they are triggered by state and rendered conditionally via a Portal. They typically contain forms or details for a single data entity. These components rely on props or context to get the data they need (for example, the requirement to edit, and a callback to save changes).

Relationships: Components primarily interact with state via props or via dispatching actions (if the state is lifted to the RequirementsDashboard reducer or a context). For instance, the Capability card in **Capabilities View** calls `handleSelectCapability` on click, which dispatches actions to update state (setting the selected capability and switching view) [8]. The RequirementsTable calls `onSearchChange` prop which in turn triggers a `dispatch` in the parent to set the search term [57].

Thus, presentational components remain mostly stateless (or have local UI state), and any action that affects global data or navigation bubbles up through callbacks to the main container.

- **State Management (Local and Shared State):** "State" refers to the dynamic data that drives the UI. In the current architecture:
- *Local component state* (`useState`) is used for ephemeral UI details (toggle flags, input values, etc.) that do not need to be known outside the component.
- *Dashboard state (useReducer in RequirementsDashboard)* acts as a UI store for global UI state (e.g., which page is showing, filters across components, whether sidebar is expanded globally). This state is local to the RequirementsDashboard component, but since that component wraps all views, it effectively behaves like a global store for the dashboard's UI context.
- *Data state via hooks:* The requirements, capabilities, and other domain data are stored within custom hooks (which themselves likely use `useState` or `useReducer` internally). For example, useRequirementsData might maintain an array of requirement objects and expose operations to alter that array. While not a global Redux store, this pattern still centralizes data management for requirements and capabilities. Multiple components can use the data from these hooks consistently as long as they use the same hook instance (in this app, the hook is called once at the top-level and its results are propagated downward). In absence of a more formal global store, the hook serves as a **controller/store** for that data domain.

Relationships: State is passed down via React props or accessed via context/hook. Components dispatch actions or call updater functions to modify state. For example, when a user edits a requirement, the **EditRequirementModal** might call an `updateRequirement` function provided by useRequirementsData; that in turn updates the requirements list state, which triggers re-render of components using that list (such as RequirementsTable or analytics charts). The RequirementsDashboard reducer state and the data hook states are kept loosely in sync through actions – e.g., selecting a capability in UI sets a filter in the reducer state, which filters the requirements data from the hook.

- **Constants:** These are static configuration objects or values. In the code, constants are defined in separate files (e.g., `constants.js`, `constants/companyProfile.js`). Their responsibility is to define **enumerations, default values, thresholds, and mappings** used by the rest of the app. For instance, status labels to color mappings, lists of compliance frameworks, thresholds for company size categories, etc., are defined as constants. Components and logic use these to avoid hardcoding values. **Relationship:** Constants are imported wherever needed. They do not hold state or interact with other parts of the app beyond providing reference data. For example, `PROGRESS_STATUSES` constant might map `"Completed"` to `{ color: "#10B981" }`, which the RequirementsTable uses to style a badge [47]. If tomorrow the color needs to change, only the constant file is updated.

- **Contexts:** Contexts provide a way to share data and functions globally without prop drilling. In this app:

- **ThemeContext** provides global theme styles and possibly theme toggling functions. Components use it to adjust class names (for dark mode vs light mode or different theme styles) [21].
- **AuthContext** provides user authentication status and information (like currentUser, roles, permissions). This can be used to conditionally render components or log actions. For example, risk creation audit logs capture the `currentUser.name` [25], which likely comes from AuthContext.

- **ToastContext** (provided by useToast hook) allows any component to show a toast message. The context stores a queue of toasts and a function to add new messages. The RequirementsDashboard calls `addToast` after certain operations (e.g. data import/export) [29] but doesn't manage the toast UI itself – that is handled by a Toast provider component globally.
- **Portal/Modal Context** ensures modals can render outside normal DOM hierarchy for accessibility; it likely provides a method to open portals and a container element.

Relationships: Context providers are usually initialized at a higher level (perhaps in App.jsx or similar) and wrap the dashboard. Components then consume context via `useContext` or custom hooks. The contexts interact minimally with each other – they tend to serve distinct purposes (theme vs auth vs toast), making them modular. For instance, the ThemeProvider might internally use the AuthContext (if, say, user preferences for theme are stored per user), but from the given code context they appear independent.

- **Controllers:** In a traditional architecture, "controllers" handle input, coordinate between model (data) and view. In this React app, there aren't explicit controller classes, but the role is fulfilled by certain functions and structures:
- The **RequirementsDashboard component** itself acts as a controller by orchestrating data and views. It listens for events (via dispatched actions or callbacks) and updates state accordingly to control what's displayed. For example, the `handleSelectCapability` function in RequirementsDashboard dispatches actions to filter requirements and switch view [8] – this is controller logic connecting the Capability UI to the Requirements data.
- The **use*Data hooks** can be seen as mini controllers for their data domain. They abstract away how data is fetched or updated. When a component calls `addRequirement`, the hook's internal logic might update state and also handle any side-effect (like saving to localStorage). This encapsulation means the component doesn't directly manipulate the data source – the hook's controller logic does.
- Event handler functions within components (like `handleCreateRisk` in RiskManagement, or `handleImportCSV` in RequirementsDashboard) also serve as controllers: they interpret user actions and perform the necessary updates. For instance, `handleUpdateRisk` in RiskManagement augments the risk object with an audit trail entry and then dispatches an action to update state [11], and similarly `handleExportNISTCSF` in RequirementsDashboard gathers assessment data and triggers a file download [28] [58]. These controllers often use utility functions to handle domain-specific calculations (e.g., adding an audit entry, calculating NIST CSF progress) and then interact with state or services.

Relationships: Controllers sit between UI and data. A controller function might use utilities (for calculations or formatting), update state (via dispatch or calling hook-provided setters), and possibly call services (if external APIs were present, controllers would initiate those calls). In current code, since external service calls are not yet integrated, controller logic remains mostly within the client (e.g., generating a UUID for a new risk, rather than requesting an ID from a backend).

- **Hooks (Custom Hooks):** The custom hooks in this project (found in the `hooks/` directory) encapsulate reusable logic tied to React's state or lifecycle. They are a bridge between pure services/utilities and the component world:
- **Data Hooks:** as discussed, manage data state and provide an interface for components to access and modify data (e.g., `useRequirementsData()` returns `{ requirements, addRequirement, updateRequirement, ... }`).
- **UI Hooks:** e.g., `usePortal` might manage creating and tearing down DOM nodes for modals, `useFilteredRequirements` computes filtered lists every time requirements or filters change (likely using `useMemo` internally for performance).

- **Analytics Hook:** `useAnalytics(requirements)` likely uses `useMemo` to derive summary statistics (like counts by category) whenever the input list changes.
- **Any External Hooks:** If the dashboard needed to integrate with an external API or event source, a custom hook would likely handle that (for example, a `useLiveThreatFeed` could subscribe to a WebSocket and provide live updates to the component).

Relationships: Hooks may depend on utilities for heavy lifting (e.g., using `transformCSVToRequirement` inside `useRequirementsData` when importing CSV data). They also may use contexts – for instance, `useCompanyProfile` might be implemented with a Context that stores the company's profile globally (so any component can get the profile and a saveProfile function) [59]. Hooks typically do not render anything themselves; they simply package logic. Multiple components can use the same hook to get shared behavior. The architecture uses hooks to enforce the **separation of concerns**: components handle rendering/UI, hooks handle data management or complex logic.

- **Services:** By "services," we refer to modules or functions that handle data operations not tied to React. In this codebase, services are often just utility modules in the `utils/` or `services/` directory:
- **Data Services:** Functions like `generateMockData`, `generateMockCapabilities` provide initial datasets. In a production scenario, these would be replaced or augmented by service calls (e.g., API fetch requests). A service layer could be created to fetch requirements from a backend, submit updates, etc. Being in a separate module, it can be reused by different parts of the app or even replaced by a different implementation (for example, a local mock vs. a live API client).
- **Computation Services:** e.g., `riskCalculation.js` might have functions to compute risk metrics; `mitreAttackMapping.js` provides functions to transform threat data structures. These are pure functions that don't depend on React and can be tested independently. They serve as the "model" logic of the application, implementing business rules.
- **External Integration Services:** Not yet present but planned (for future API integration). One could envision a file like `apiService.js` or similar where all HTTP requests are made (using `fetch` or Axios). The hooks would then call these service functions to get data, rather than generating mock data.

Relationships: Services are utilized by hooks or controller functions. They do not directly interact with React components or contexts; instead, they provide results that controllers insert into state. This decoupling ensures that if the data source changes (from mock to real API), the component code does not need major changes – only the service layer or the hooks using them would change.

- **Stores:** In the context of this app, "stores" could refer to centralized places where state lives (in a broader sense than React's local state). There isn't a Redux or MobX store here, but we can consider:
- The **RequirementsDashboard reducer state** as a UI store.
- The **data managed by hooks** as domain stores (for requirements, capabilities, etc.). If these hooks were refactored to use React Context, each could become a small Redux-like store for that part of the data, accessible via context to any component.
- **LocalStorage or browser storage** might act as a persistence store. For example, the code snippet suggests localStorage is used for persisting standards assessment data across sessions (the useEffect saving NIST CSF data was hinted in code). If so, on initialization, the hooks or components might load data from localStorage (acting as a store that outlives page refresh).

Relationships: Stores (or state containers) feed the components with data. The components dispatch actions or call functions to change the store, which then notifies all consumers (causing re-renders with updated data). Currently, the store concept is mostly confined within the component/hook that owns

the state. A redesign might externalize these stores using Context providers (turning them into more explicit stores accessible app-wide).

- **Styles:** Styling in the dashboard is primarily done via utility classes (looks like Tailwind CSS classes) embedded in JSX (`className="bg-white rounded-xl shadow-md p-6"`, etc.). The presence of classes like `text-gray-900`, `px-4 py-2`, and responsive utilities (`lg:flex-row`, `xl:grid-cols-2`) strongly indicate TailwindCSS usage. The ThemeContext may apply additional classes or CSS variables for theme switching:
- e.g., for a "stripe" theme, it uses classes like `main-background`, `sidebar-gradient`, which are likely defined in a global CSS file or Tailwind config to produce a striped background and a gradient sidebar [60].
- For the default theme, it falls back to plain colors (`bg-gray-50` for background, `bg-gray-900` for sidebar, etc.) [61].

Additionally, the application likely has a global stylesheet for some base styles (perhaps Tailwind's base and components, plus custom styles for classes like `.line-clamp-2` used to truncate text in capability description [62]). There might also be SCSS or CSS modules for specific components, but given the heavy use of inline utility classes, custom CSS usage seems minimal and mainly through theme classes.

Relationships: The style system is connected to the Theme context (to dynamically change themes) and to the design system. If atomic design were applied, styles would be systematically managed at the atomic (small component) level and propagate consistently. Currently, consistency is maintained by reusing classes and the occasional function for style (e.g., `getStatusStyle()` in RequirementsTable centralizes status badge CSS classes [63], and similar for priority and applicability [64] [65]). This is a form of style utility to ensure consistency across the app. The style approach is largely **framework-agnostic** (just CSS classes) and doesn't lock the project into a specific styling library beyond Tailwind (which can be swapped or extended as needed).

- **Utilities:** These are standalone helper functions that do not maintain state or produce UI, but perform calculations, format data, or generate structures used by the rest of the app. They serve as the **toolbelt** of the application logic:
- They **promote reuse** (e.g., one function to validate an email format can be used wherever needed in the app).
- They **simplify components and hooks** by offloading complex calculations (e.g., determining risk severity from a score is done in `getThreatLevel` utility [54] rather than recalculated in multiple places).
- They can be **tested independently**. For instance, `calculateRiskScore(probability, impact)` returns a number and can be unit tested with various inputs to ensure correctness [53].

Relationships: Utilities are the lowest-level layer, with no knowledge of React or state. They are called by controllers, hooks, or occasionally directly in components for simple tasks. For example, when rendering a list of frameworks in the UI, the component might call `getSuggestedFrameworks(companySize)` to get an array of frameworks to display [49], keeping the component code very clean. Utilities often work in tandem with constants (e.g., the thresholds constant is used inside a utility function to determine company size category [66]).

In summary, the current architecture is somewhat layered but with a few **blurring of boundaries**: the RequirementsDashboard plays multiple roles (view, state store, and controller), and the lack of a dedicated global state/store means the component carries a heavy load. The relationships between

these categories are mostly managed through React's composition and Context. For example, components rely on hooks (that use services and utils) to get data; the main component uses reducer state to coordinate UI across sub-components; constants and utils feed into both hooks and components where needed.

Pain Points in the Current Architecture

While the application is functional and integrates many features, several architectural pain points and antipatterns are evident:

- **Monolithic Main Component:** The **RequirementsDashboard.jsx** file is extremely large (thousands of lines) and manages a wide breadth of functionality. This violates the single-responsibility principle – it acts as a page layout, a state manager, and a controller for various features. Maintaining or extending this file is difficult; any change risks side effects across unrelated sections. For instance, it contains logic for handling keyboard shortcuts, window resize events, data import/export, and conditional rendering of over ten different sub-views all in one place. This complexity makes the component hard to understand and **not easily scalable** if new sections are added.

- **Tight Coupling of Concerns:** UI logic, state management, and business logic are intermixed in the same places. For example, consider how selecting a capability is handled: the click event triggers `handleSelectCapability` (UI control logic) which directly dispatches state changes to filter data and change the view [8]. The filtering of requirements by capability is thus tied to the act of view navigation. Ideally, selecting a capability could simply set a route or context state, and a separate effect could manage applying filters. In the current setup, one action performs multiple tasks. Similarly, RequirementsDashboard's reducer manages both UI state (like modals) and holds some derived data (like selectedCapability in filters). This coupling means that a change in data structure can affect UI reducer logic and vice versa, leading to **tight interdependencies**.

- **Lack of Modularization in State:** The application has multiple domains of data (requirements, capabilities, threats, risks, etc.), but there is no clear separation of state for each domain. Instead, the main reducer state and a few hooks hold everything. For example, risk management uses a separate reducer inside its component, but requirements and capabilities rely on hooks while standards (NIST CSF data) are stored in the main reducer. There's an inconsistency in how state is handled, which can be confusing. Also, since the main reducer's state shape is large, passing needed parts to sub-components can result in prop drilling or accessing via closures. **Scaling this state** becomes problematic – adding a new global piece of state means touching the big reducer and context around it.

- **Implicit Global State (via Hooks):** The custom data hooks provide a form of global state, but this is not immediately obvious. For instance, `useRequirementsData` might internally use `useEffect` to load data from localStorage and keep it in a module variable. If multiple components were to call `useRequirementsData` separately, would they share the same data or get duplicates? Likely it's designed to be called once in the main dashboard. This pattern works but is not the standard way to share state in React; typically one would use Context or an external store so that data is truly singleton. Without clear documentation, a new developer might mistakenly call `useRequirementsData` in two different components and then wonder why actions in one don't affect the other. This is a **pitfall of using hooks as a store** without context – it relies on usage conventions rather than enforceable architecture.

- **No Clear Separation Between Framework and Application Logic:** Much of the logic is tightly bound to React (through hooks, component lifecycle, and state). For example, the NIST CSF assessment calculations are defined inside the component as functions and state [67] [68], rather than being moved out to a pure utility or service. This means those computations are re-created on each render and can't be easily reused elsewhere or in non-React contexts. If in the future a different frontend or a backend service needed to use the same calculation, one would have to duplicate the logic. This indicates a **lack of portability** in some parts of the code.

- **Incomplete Atomic Design Implementation:** The app does have many small reusable components (atoms like buttons and icons from a library, small UI components like **MaturityIndicator**, and larger composites like **RequirementsTable** which can be seen as an organism). However, these components are not structured in a formal atomic design system. The file structure seems to be grouped by feature rather than by atomic levels. For example, **StatCard** and **MaturityIndicator** live in a `components/ui` folder, and feature-specific components live in `components/threats`, `components/requirements`, etc. This is a feature-based organization, which is fine, but within those, the atomic concept (atoms/molecules) isn't explicitly delineated. Also, some "atoms" are not abstracted: e.g., the code duplicates certain UI patterns like styled section headers or repeated card layouts (capability cards, risk cards) rather than abstracting them into a single Card molecule. This leads to **inconsistencies** or repeated code, and makes it harder to globally update UI style since changes might need to be applied in multiple places.

- **Hardcoded Mock Data & Missing API Layer:** Currently, the data is generated via mocks (e.g., `generateMockRisks()` in RiskManagement [69], mock requirements/capabilities in dataService). There's no abstraction for data fetching – the logic is embedded in the app. When it's time to integrate external APIs, the current architecture will require significant changes. For instance, an `addRequirement` function might directly update local state and maybe localStorage, but in a real scenario it needs to call an API and handle async responses, errors, etc. There is no provision for loading states or error states at the data-fetch level (aside from a simple `loading` boolean possibly provided by hooks). **Pain point:** introduction of real data sources could be messy, because the app wasn't clearly structured with a service layer and asynchronous flow in mind. We see a hint of an `error` state from useRequirementsData [70] and how the component shows an error screen if `error` is truthy [71] [72]. But error boundaries and loading placeholders exist only at a high level; a more granular approach might be needed when each piece of data can load/fail independently.

- **Navigation and Extensibility:** The dashboard uses an internal state to switch views rather than a routing mechanism. This works for a contained app but can become a pain point if deep linking or browser navigation (back/forward) is needed. Without using something like React Router, the app cannot be easily linked to a specific sub-view or requirement detail – it's all one page. As the application grows (e.g., adding more distinct pages or wanting to integrate it with other web applications), this single-page state machine may become cumbersome. It also means that all components for all views are imported into this one file, potentially increasing the initial bundle size and load time (though some components might be lazy-loaded; the code shows `lazy` imports for some, but it's minimal in the snippet). Managing such global navigation state manually is error-prone – the keyboard shortcuts logic mapping numbers to view names is one example that needs updating whenever views are added or reordered [73]. This is a **maintenance risk**.

- **Duplication of Logic / Inconsistent Patterns:** Some features seem to have been added incrementally, leading to slight inconsistencies. For instance, risk management is encapsulated

in its own component with its own state, whereas requirements management is partly in the main component and partly in a child component. The standards (NIST CSF) logic is partly stored in state and partly handled via inline component (the NISTCSFAssessment component defined inside RequirementsDashboard). This inconsistency can confuse the architecture: why isn't NISTCSFAssessment its own module? Why does risk management not integrate with the main state? These inconsistencies suggest **architecture erosion**, where initial patterns weren't uniformly applied as new features were added.

- **Scalability Limits:** If the number of requirements or capabilities grows large, the current approach might face performance issues. For example, filtering all requirements on every render (depending on how useFilteredRequirements is implemented) could become slow if not optimized. There's no mention of virtualization for long lists (if requirements become hundreds or thousands, a simple table might lag). The all-in-memory approach will also strain the browser for very large data sets. Without an architecture that supports gradual data loading (pagination, on-demand fetch), scalability in terms of data volume is limited.

In summary, the architecture's main pain points are about **separation of concerns and scalability**. The code works for a prototype with a moderate amount of data and features, but as features and data grow, the current structure would become unwieldy. Key issues to address include breaking down the monolith, clearly separating UI from logic, preparing for external data sources, and organizing the code in a more systematic, maintainable way.

Recommendations for Dashboard Redesign

To improve the modularity, scalability, and clarity of the Cyber Trust Sensor Dashboard, a thorough refactoring and re-architecture is advised. The following best practices and design recommendations will address the pain points:

1. Adopt a Modular, Feature-Oriented Architecture

Restructure the application by grouping related functionality into modules or feature domains. Each module can encapsulate its own components, state management, hooks, and services. For example: - **Requirements Module:** Contains components (RequirementTable, RequirementItem, etc.), a context or store for requirements state, and services for requirement data (API calls or local operations). - **Capabilities Module:** Manages capability-related components (CapabilityList, CapabilityCard, CapabilityDetail), state (selected capability, etc.), and any capability-specific logic. - **Risk Management Module:** Already somewhat separated, ensure it has its own directory with components, reducer, and possibly context provider so it can be loaded or used independently. - **Threat Intelligence Module:** Components like MitreAttackNavigator and ThreatIntelligenceSystem plus related utilities (e.g., MITRE mappings) reside here. - **Profile/Settings Module:** CompanyProfileSystem and SystemSettings components, along with profile context and utilities, in their own section. - **Shared Module:** Cross-cutting components (like the UI library components – buttons, modals, cards) and hooks (theme, auth, toast) can live in a common area.

This modularization makes the codebase more understandable. New developers can navigate the project by feature, and future additions (e.g., adding a new module for "Audit Logs" or "User Management") can be done without touching core files of other modules. It also limits the impact of changes – changes in one module ideally shouldn't cascade into others if the interfaces between modules are well-defined.

2. Introduce an Atomic Design System

Implement an **Atomic Design** methodology to organize and build the UI systematically. Atomic Design breaks the UI into five levels: **Atoms, Molecules, Organisms, Templates, Pages** [74]. In practice: - **Atoms:** Basic UI elements that can't be simplified further – for instance, a button, an input field, an icon, a label, etc. These should be reusable across the app. In code, atoms might correspond to simple styled components or wrappers around HTML elements (e.g., a custom `<Button>` component that already has the default classes for styling, so you don't repeat `className="px-4 py-2 ..."` everywhere). - **Molecules:** Small combinations of atoms that form a unit with a specific function. For example, a search bar molecule might combine an input atom and a button atom. In this dashboard, a table row component showing requirement details (with status badge, text fields, icons) could be a molecule composed of multiple atoms. - **Organisms:** Larger sections of the UI, assembling multiple molecules and/or atoms. A table component (with header, rows, pagination controls) can be an organism. A modal dialog with a form is another organism. The Capability Card shown in the grid is effectively an organism: it contains multiple text elements (atoms) and icons, possibly some molecules for metrics. - **Templates:** Page layouts that arrange organisms into a full page structure, without actual dynamic content. A template defines regions (header, sidebar, content area, footer) and placeholders for organisms. For the dashboard, you might have a Dashboard template that puts a navigation sidebar next to a content area; within the content area, different organisms appear depending on the page. - **Pages:** Realized templates with actual content/data. The Requirements Dashboard page is a concrete page where the template is filled with actual data (list of requirements, etc.).

By using atomic design, the team can ensure consistency and reusability: - Create a **UI component library** (atoms/molecules) that can be documented and possibly tested in isolation (using Storybook, for example). This would include things like `Button`, `Card`, `Modal`, `FormField`, `Badge` (for status labels), etc., as well as composite molecules like `SearchBar`, `StatsGroup` (maybe the row of StatCards at the top of overview). - Use those consistently across all features. This reduces CSS class repetition and ensures that updating the style of an atom (say, changing the primary button color) propagates everywhere. - The design system can also enforce a unified look and feel which is important for scaling (especially if multiple developers work on the UI, atomic components act as a source of truth).

If the user is unfamiliar with atomic design, in short: **Atomic design** is a methodology for creating design systems by building up from simple elements to complex pages [74]. We start with small components (atoms), combine them into slightly larger components (molecules), then assemble those into full sections (organisms), which are placed in page layouts (templates), and finally filled with content (pages) [75] [76]. This approach encourages reuse and single responsibility at each level [77], leading to UIs that are easier to maintain and extend.

3. Strengthen Separation of Concerns

Refactor the code so that **each piece has a single responsibility** and minimize direct coupling: - **State vs UI:** Migrate away much of the UI state currently in RequirementsDashboard's reducer into either local state (if truly local, e.g., show/hide column selector can remain in RequirementsTable) or into dedicated contexts/stores for broader state. For example, have a **RequirementsContext** with its own reducer that tracks filters, search term, etc., related to requirements. The main dashboard would then simply render the Requirements view (component) within a provider of RequirementsContext. The Requirements components would consume that context. This way, if you use the Requirements components elsewhere or independently, you just wrap the context and they work – they are not hardwired to the giant dashboard component. - **UI State vs Data State:** Distinguish between *UI state* (view modes, which modal is open, layout toggles) and *Data state* (the actual list of requirements, etc.).

They can be managed separately. UI state can remain in a main UI context or even a simple useState in a top-level component. Data state should perhaps reside in a global store or be fetched on demand. By decoupling these, you could, for example, reuse the data store in a different UI (or multiple UIs) or swap out the UI without touching how data is handled. - **Move Business Logic Out of Components:** Any significant computation or data transformation (like calculating completion rates, scores, filtering lists) should reside in either **utility functions** or within the state management layer (e.g., a selector function in a Redux-style store or a function returned by a hook). Components should ideally just call a function to get derived data and render it. Currently, we see computations inline (e.g., calculating `completionRate` for each capability within the render map [78]). That logic can be moved to a helper (e.g., `getCapabilityCompletion(capabilityId)` that returns completed vs total counts). This makes it easier to test and also to optimize (the helper could memoize results or a context could provide those precomputed values). It also keeps render functions cleaner. - **Define Clear Interfaces Between Modules:** Modules/contexts should expose well-defined interfaces. For instance, the RequirementsContext might expose a context value like `{ requirements, filteredRequirements, filters, setFilter, addRequirement, ... }`. Components using it don't need to know how it's implemented (whether it's Redux, useReducer, XState, etc.). This abstraction allows you to change the underlying mechanism without affecting components. It also means one module could be completely replaced (say you want to use a different library for state management or you split the app into microfrontends) with minimal changes to others, as long as the interface remains consistent. - **Use React Router (or Routing)** for Navigation: Instead of managing `viewMode` in state with manual toggles, consider using a router. Each major view (requirements, capabilities, risk management, etc.) can be a route (e.g., `/dashboard/requirements`, `/dashboard/capabilities`). This provides deep linking, back button support, and a clearer separation – each route can have its own data loading logic or context providers. For example, the RiskManagement component could be mounted only when the `/dashboard/risk-management` route is active. You can still keep a side navigation that uses router links. This change would remove a lot of conditional rendering logic and simplify the RequirementsDashboard (which might just become a wrapper with a Switch of routes inside). It also naturally lazy-loads code for other routes (with `React.lazy` and Suspense or code-splitting), improving performance. - **Smaller Components and Functions:** Break down large functions and components. If a component file exceeds a few hundred lines, it likely does too much. For instance, the NIST CSF assessment part defined inside RequirementsDashboard should be a separate component file (e.g., `NistCsfAssessment.jsx`) – it can still use the state via props or context, but isolating it improves readability and maintainability. The reducer actions related to standards could also be isolated in their own context or reducer, so that the main dashboard reducer doesn't have to handle those cases.

4. Prepare for External API Integration

To integrate external APIs in the future, introduce a **service layer** abstraction now: - **API Service Modules:** Create modules to handle API calls, one per resource or feature. For example, `api/requirements.js` might export functions like `fetchRequirements`, `createRequirement`, `updateRequirement`, etc. Initially, these can call local mock functions or return promises that resolve mock data. Later, they can be easily switched to do real `fetch`/Axios calls to a backend. By centralizing API interactions, you avoid sprinkling `fetch` calls throughout components, which can be hard to refactor. It also centralizes error handling – e.g., an API service function can catch errors and standardize them (maybe converting various HTTP errors into a consistent error object). - **Asynchronous Flow Handling:** Modify the data hooks or state management to handle loading and errors more robustly. Instead of assuming instant availability of data (as mocks do), have states: e.g., `requirementsLoading`, `requirementsError`. The UI should check these and render loading spinners or messages (some of this is already done at a global level [79] [71], but it might need to be more granular if different parts of the page load at different times). - **Decouple Data Generation from Components:** Right now, data generation (like `generateMockData`) is called likely within the hook or

component. In a redesign, the decision of using mock data vs real API could be toggled via a configuration. For instance, if no backend is available, the `fetchRequirements` service could internally call `generateMockData` and return that; if a backend exists, it does a network request. The rest of the app just calls `fetchRequirements`. This strategy allows development and testing to use local mocks seamlessly and switch to real data in production or staging. It also supports a future where the app might allow plugging in different data sources (making it somewhat "framework-agnostic" in terms of data backend). - **Batch and Cache Data:** In preparation for real APIs, consider how data is shared. If one view needs to show a count of all requirements and another view shows the list of requirements, avoid fetching the same data twice. A centralized store or context for requirements can load data once (when the app or that route initializes) and provide it to all components. Perhaps use something like React Query or SWR (if suitable) to handle caching and background refreshing of data from APIs. This will ensure the app is efficient and reduces duplicated calls.

5. Ensure Framework-Agnostic Extensibility

The user mentioned being framework-agnostic for future "bolt-ons" – this likely means making sure the design is not so tightly tied to React that other systems can't interface with it. Possible interpretations and solutions: - **Micro-frontend readiness:** If future bolt-ons might be micro-frontends or plugins possibly written in other frameworks, the architecture should isolate each feature's code. By using routing and context boundaries, one could potentially swap out a route's implementation entirely. For example, the threat intelligence module could in the future be served by a different team as a micro-frontend. If our app is modular, it's easier to integrate that – essentially treating that part as an external component. - **Use Standard Interfaces:** Ensure that communication between modules or with external plugins uses standard web technologies (e.g., events, URL query parameters, or a shared store via something like Redux that could be interfaced with externally). Avoid highly React-specific coupling for things that might be shared. For instance, if an external script needs to update something in the dashboard, providing a global event or using the browser's Storage events might be more framework-agnostic than expecting it to import our React context. - **Decouple Styling and State from React where possible:** If "framework-agnostic" implies that parts of the system might be used outside React, one strategy is to keep core logic in plain TypeScript classes or functions. For instance, have a class `RequirementsManager` that contains all logic to manage requirements (add, edit, compute stats) not relying on React. The React components/hooks then use an instance of this class (maybe via Context) to manipulate data. If one day a non-React component needs to manipulate requirements, it can use the same class. This essentially follows an MVC approach where React is only the View, and the Model/Controller could be in plain JS. This might be over-engineering for now, but it's a thought for high extensibility.

- **Documentation and Contracts:** Define clear integration points and document them. If external systems will interact, maybe define a plugin interface or API (even if it's just posting messages to a window, etc.). While this is outside pure code architecture, it's part of planning for future extensibility.

6. Proposed Folder and File Structure

Reorganizing the project's folders will greatly enhance maintainability. Here is a suggested structure following a feature-modular and atomic design approach:

```
src/
├── components/  (or 'ui/')
│   ├── atoms/
```

```
│   │   ├── Button.jsx
│   │   ├── Input.jsx
│   │   ├── Icon.jsx  (wrapper for lucide-react icons with common props)
│   │   ├── Badge.jsx (for status, priority labels styled by type)
│   │   └── ... (other basic elements)
│   ├── molecules/
│   │   ├── SearchBar.jsx  (uses Input + Button atoms)
│   │   ├── StatsCard.jsx  (possibly what StatCard is, combining Icon + some
text)
│   │   ├── RequirementRow.jsx (a row in the requirements table as a separate
component)
│   │   └── ...
│   ├── organisms/
│   │   ├── RequirementsTable.jsx
│   │   ├── CapabilityCard.jsx
│   │   ├── RiskList.jsx (list of risks, using molecules like RiskItem)
│   │   ├── Header.jsx (top header bar for the dashboard)
│   │   ├── Sidebar.jsx (navigation sidebar)
│   │   └── ... (other larger composite components)
│   ├── templates/
│   │   ├── DashboardLayout.jsx (puts Header, Sidebar, Content region)
│   │   ├── ModalLayout.jsx (common layout for modals perhaps)
│   │   └── ...
│   └── pages/
│       ├── DashboardPage.jsx (uses DashboardLayout and includes perhaps
overview)
│       ├── RequirementsPage.jsx (includes RequirementsTable in layout)
│       ├── CapabilitiesPage.jsx
│       ├── RiskManagementPage.jsx
│       ├── ThreatIntelligencePage.jsx
│       ├── StandardsPage.jsx
│       └── ... (each route gets a page component)
├── features/  (feature modules grouping context, hooks, services)
│   ├── requirements/
│   │   ├── RequirementsContext.js (provide reducer or zustand store for
requirements)
│   │   ├── requirementsReducer.js (if using useReducer)
│   │   ├── useRequirementsData.js (if still needed, or integrated into
context)
│   │   ├── RequirementsService.js (API calls for requirements)
│   │   └── RequirementModel.js (optional: class or TS type definitions)
│   ├── capabilities/
│   │   ├── CapabilitiesContext.js
│   │   ├── capabilitiesReducer.js
│   │   ├── CapabilitiesService.js
│   │   └── ...
│   ├── risk/
│   │   ├── RiskContext.js
│   │   ├── riskReducer.js
│   │   ├── RiskService.js
```

```
│   │   └── ...
│   ├── threats/
│   │   ├── ThreatContext.js (if needed)
│   │   ├── ThreatService.js
│   │   ├── mitreAttackMapping.js (could be here or in utils)
│   │   └── ...
│   ├── profile/
│   │   ├── ProfileContext.js
│   │   ├── ProfileService.js
│   │   └── ...
│   └── analytics/  (analytics might not need separate context, could just be
util)
│       └── analyticsUtils.js
├── services/  (if not grouped by feature, common services could live here)
│   ├── apiClient.js (setup of axios or fetch wrapper)
│   ├── requirementsApi.js
│   ├── capabilitiesApi.js
│   ├── riskApi.js
│   └── ...
├── utils/
│   ├── csvUtils.js
│   ├── formatUtils.js (e.g., date formatting)
│   ├── calcUtils.js (common calculations)
│   ├── constants.js (general constants or export from sub-files)
│   ├── constants/
│   │   ├── companyProfile.js
│   │   ├── threatConstants.js
│   │   └── ...
│   └── ... (any other pure utility modules)
├── contexts/  (app-wide context providers)
│   ├── ThemeProvider.jsx
│   ├── AuthProvider.jsx
│   ├── ToastProvider.jsx
│   └── maybe GlobalStateProvider.jsx if using a combined provider
├── styles/
│   ├── tailwind.css or tailwind.config.js (if using Tailwind)
│   ├── theme.css (for custom theme classes like .main-background, .sidebar-
gradient)
│   └── ... (any global styles or CSS resets)
└── App.jsx (setup providers, routes using pages, etc.)
```

Note: The above structure is one of several possibilities, but it demonstrates separating concerns: - UI components by atomic levels, - Feature-specific logic in their own areas, - Shared contexts and utilities in accessible places, - Clear entry points for API interactions.

Using this structure: - A developer working on the Requirements feature knows to look under `features/requirements` for business logic and under `components/organisms/RequirementsTable.jsx` for the UI. - The atomic components in `components/atoms` and `molecules` would be used by multiple features, promoting consistency (e.g., a `Badge` atom for

statuses can be used in Requirements table, Capability cards, Risk items, etc.). - Adding a new page is straightforward: create a new Page component (perhaps using existing organisms or new ones) and add a route in App.jsx. - If the project grows, one could even turn each feature module into an npm package or a microfrontend boundary with relatively low coupling between them (since interactions go through context or service interfaces).

7. Reusable Utilities and Shared Context Patterns

To avoid duplication and foster reuse, identify common patterns that can be abstracted: - **Form Handling Utilities:** If forms are used in multiple modals (e.g., edit requirement, edit risk, profile form), consider a form utility or a custom hook like `useForm` that manages form state and validation. Common validations (like required fields, email format) can be in utils so that all forms are consistent in feedback. - **Table/List Patterns:** RequirementsTable and other lists (risks, possibly capabilities) share features like filtering, sorting, searching, pagination. A reusable list component or hook could handle these concerns generically. For instance, a `useFilterSort` hook that takes in a dataset and filter criteria, and returns a filtered list and actions to update filters. - **Modal Pattern:** Use a shared Modal component (already exists) with a consistent style. Ensure all modals follow the same pattern for close behavior (perhaps ESC key to close is handled by a single Modal wrapper, rather than each modal adding its own event listener as currently in RequirementsDashboard's effect [80]). This can be achieved by a global modal context or by enhancing the Modal component to handle it internally. - **Toast Notifications:** The toast system is already centralized. Continue using a single ToastProvider and ensure any new feature just calls `addToast` without needing its own implementation. - **Context Composition:** Rather than having one giant context or many unrelated contexts at the root, use context composition wisely. For example, within the Requirements page, use the RequirementsContext provider. Within the RiskManagement page, use the RiskContext provider. They only live when that part of the UI is active, which is good for performance and memory. For truly global things (current user, theme), keep them at the top. This pattern avoids global namespace pollution and keeps contexts focused. It also means if one part of the app is removed or replaced, its context can go with it without affecting others. - **Custom Hook Libraries:** Create hooks that can be reused across features: - `useDebouncedSearch(term, delay)` – could be used anywhere a search input is present to debounce input changes. - `useLocalStorage(key, defaultValue)` – a hook to persist a piece of state to localStorage (e.g., preserving filters or user settings between sessions). - `useToggle(initialValue)` – simple hook to get a boolean and a function to toggle it (useful for show/hide toggles). These small hooks reduce boilerplate and ensure consistent behavior across components (for example, every debounced search will behave the same).

- **Shared Data Context When Needed:** If requirements and capabilities are closely related (they are – a requirement belongs to a capability), consider if a combined context makes sense or if separate contexts that reference each other. Perhaps a **ProjectContext** that provides both requirements and capabilities together (since they are often used in tandem) could simplify things – for instance, adding a new capability might implicitly update the requirements filter options, etc. However, this should be designed carefully to avoid tangling unrelated data – if capabilities and requirements are mostly handled separately except for the linking field, separate contexts with clear interaction (like calling a method in requirements context to filter by a capability ID) is fine.

8. Requirement Data Handling and Traceability

In the current system, requirements belong to capabilities via a `capabilityId` field. There is a potential need to handle **shared requirements across capabilities**, meaning a single requirement can be linked to multiple capability areas (or perhaps multiple standards). To make this traceable and

maintainable: - **Use Unique IDs and Reference Lists:** Instead of a single `capabilityId` per requirement, allow a requirement to have an array of associated capability IDs if the data model permits. This way, one requirement object is not duplicated; it just has references to multiple capabilities. In a relational sense, this is a many-to-many relationship which could be managed via a join table or in code by maintaining a mapping (e.g., a dictionary where key is capability and value is list of requirement IDs, and vice versa). - **Centralize Requirement Store:** Keep all requirements in one collection (e.g., in RequirementsContext). If a requirement is shared, store it once. In each Capability's context or data, store only the IDs of requirements it includes. This prevents divergence (where editing a "shared" requirement in one place and forgetting to update in another). - **Traceability Features:** Implement mechanisms to easily trace where a requirement is used: - On the UI, if a requirement is shared by multiple capabilities, the requirement detail view could list all capabilities (or standards) that reference it. - Internally, utilities can be provided to query this. For example, `getRequirementCapabilities(reqId)` would return all capabilities that include that requirement. If using contexts, the CapabilitiesContext could have a selector to do this by scanning its data structure. - **Consistency on Edit/Delete:** Ensure that when a shared requirement is edited, the change reflects for all capabilities. This naturally follows if there's a single source of truth for the requirement. Deleting a shared requirement might need confirmation if it affects multiple places, or alternatively, a "removal" from one capability might just unlink it rather than delete entirely, depending on desired behavior. These rules should be clearly defined and the code structured to enforce them (possibly within the RequirementsService or context actions). - **Versioning (if needed):** In compliance-heavy environments, sometimes you need to track changes over time. If that is a future concern, designing the data model to accommodate version IDs or timestamps on requirement updates could be useful. It's beyond the current scope, but good to keep in mind when refactoring the data handling (e.g., maybe a requirement has a history log or status transitions that need to be preserved).

- **Maintainability:** Write clear documentation or code comments on the relationships. If someone sees a `requirements` array and a `capabilities` array, they should be aware of how they link. One could implement TypeScript interfaces or PropTypes to indicate that a requirement can have multiple links. High cohesion in code (like grouping related actions together in reducer or using TypeScript to enforce linking logic) will reduce errors.

By addressing requirement data management with these approaches, the dashboard will be ready to handle more complex relationships without data duplication or loss of integrity. It will also help in generating reports or analytics that consider shared requirements (for instance, "how many unique requirements do we have vs. how many assignments of requirements across capabilities").

9. Visual Aids

(The following visualizations provide a high-level understanding of the proposed architecture.)

Proposed Folder Structure: The project's source tree could be organized as follows for clarity and scalability:

```
src/
├── components/           # Atomic Design components library
│   ├── atoms/            # Basic UI elements
│   ├── molecules/        # Small combinations of atoms
│   ├── organisms/        # Complex components (sections of UI)
│   ├── templates/        # Page layouts
│   └── pages/            # Complete pages tying everything together
```

```
├── features/              # Feature-specific logic and state
│   ├── requirements/      # Requirements domain (state, hooks, services)
│   ├── capabilities/      # Capabilities domain
│   ├── risk/              # Risk management domain
│   ├── threats/           # Threat intelligence domain
│   └── profile/           # Company profile and settings domain
├── contexts/              # Application-wide contexts (Theme, Auth, Toast,
etc.)
├── services/              # API interaction modules (if not within feature
directories)
├── utils/                 # Utility functions and constants
└── styles/                # Global styles (Tailwind CSS, theme CSS)
```

Each feature directory may contain its own context provider and reducer for that feature's state, plus any API service functions and specialized utilities. The atomic components in `components/` are used across these features to build consistent UIs.

Data Flow Diagram (simplified): When the user interacts with the dashboard: 1. **UI Layer (Component)** – e.g., user clicks "New Capability" button (an atom/molecule component). 2. **Controller/ Handler** – the onClick event triggers a function in the Capability context (or dispatches a `SHOW_NEW_CAP_MODAL` action to the UI store). 3. **State Update** – The context or reducer updates the state (e.g., sets `showNewCapabilityModal = true`). This re-renders components that depend on this state. 4. **Modal Opens (UI)** – The NewCapabilityModal organism appears. The user fills the form and submits. 5. **Service Call** – A submit handler in the modal's component calls `CapabilitiesService.createCapability(data)` (could be directly or via context dispatch). 6. **Data Layer** – The service sends an API request to the backend (or calls a mock function). On success, the returned new capability is passed to the CapabilitiesContext. 7. **State Update** – CapabilitiesContext reducer adds the new capability to its list. If Requirements are affected (maybe new capability has no requirements yet, but the filter lists could update), those contexts might remain unaffected or could have side-effects (if needed, e.g., adding a new filter option). 8. **UI Refresh** – The capability list organism re-renders with the new capability visible. The modal closes (state toggled off). 9. **Toast Notification** – Using ToastContext, a success message "Capability created" is shown.

Throughout this flow, each part (UI, context, service) has a distinct role, making the logic easier to trace and debug. Logging can be added at service or context layer to audit actions (useful for debugging or compliance).

By implementing these recommendations, the Cyber Trust Sensor Dashboard will become more **modular, easier to maintain, and scalable** for future needs. The adoption of atomic design and clear separation will enhance consistency in the UI/UX, while the improved state management and service layers will make the app robust in the face of growing data and integration requirements. Moreover, the system will be better prepared to integrate external modules or APIs and to support extensions without breaking the core architecture.

Next Steps: We propose a phased project plan (see CSV below) to execute this refactoring and enhancement process in stages, ensuring that at each phase the system remains functional and testable.

Project Plan

Below is a high-level project plan broken into phases with tasks, priorities, dependencies, time estimates, and responsible parties:

```
Task,Description,Priority,Dependency,Estimated Time (days),Responsible Party
Architecture Audit,"Review current codebase, identify all components, state and data flows in detail",High,None,2,Lead Developer
Requirements Gathering,"Confirm new architecture goals (atomic design adoption, API integration, etc.) with stakeholders",High,None,1,Project Manager
Design New Architecture,"Create detailed design docs for new structure, contexts, and services",High,Architecture Audit,3,Software Architect
Set Up Atomic Design Framework,"Establish atoms, molecules, organisms structure and create initial reusable components (buttons, inputs, modals, etc.)",High,Design New Architecture,4,UI Developer
Implement Core Contexts,"Create Theme, Auth, Toast providers (if not already) and new global or feature contexts for Requirements, Capabilities, etc.",High,Design New Architecture,3,Lead Developer
Refactor Data Hooks to Services,"Introduce service modules and update hooks/contexts to use API services (with mock data initially)",High,Design New Architecture,4,Backend Developer
Module Separation,"Move code into new feature modules (requirements, capabilities, risk, etc.), implement context providers and reducers for each",High,Core Contexts,5,Lead Developer
Routing Implementation,"Integrate React Router (or alternative) for distinct views, replace viewMode state with routes",Medium,Module Separation,2,Frontend Developer
Refactor RequirementsDashboard,"Break apart monolithic component into Page components and Layout; use contexts and routing to simplify",Medium,Module Separation,4,Frontend Developer
Integrate Atomic Components,"Replace old UI elements in components with new atoms/molecules (ensure visual consistency)",Medium,Set Up Atomic Design Framework,5,UI Developer
Shared Requirement Handling,"Modify data model to support requirements linked to multiple capabilities; update context logic and UI to reflect this",Medium,Module Separation,3,Backend Developer
Testing Phase 1,"Test core functionality after restructure (smoke test major features, ensure no regressions in data display)",High,Module Separation,3,QATester
UI Redesign/Polish,"Apply new styling guidelines, ensure responsive design still works, improve any UX flows as needed",Medium,Integrate Atomic Components,4,UI Designer
Performance Improvements,"Add optimizations (lazy loading, list virtualization for tables, etc.) where needed for scalability",Low,Testing Phase 1,3,Frontend Developer
API Enablement,"Swap mock services with real API calls, implement API client and error handling, adjust contexts for async",High,Refactor Data Hooks to Services,5,Backend Developer
Testing Phase 2,"Test with API integration (test all CRUD operations, error
```

```
states, authentication flows)",High,API Enablement,4,QATester
Documentation & Training,"Update README, architecture docs, and conduct a
walkthrough for the team on the new structure",Medium,Testing Phase
2,2,Software Architect
Deployment & Monitoring,"Deploy updated application, monitor logs and
performance, ensure stability in production",High,Testing Phase 2,1,DevOps
Engineer
```

---

1 2 3 6 7 8 9 10 14 15 16 17 18 19 20 21 22 23 24 27 28 29 30 32 33 34 35 36 37 38 39 40 41 42 51 52 57 58 59 60 61 62 67 68 70 71 72 73 78 79 80 RequirementsDashboard.jsx

file://file-72UbX3Nko8QYkqU3JcghdX

4 5 31 46 47 63 64 65 RequirementsTable.jsx

file://file-Qk9tESYAencwcMqgWAhEoS

11 12 13 25 26 69 RiskManagement.jsx

file://file-Y6gmTSwNpWb5yuBXyyq8KW

43 48 49 50 66 companyProfile.js

file://file-6zd93EPz7eN9EzYjrcLpUY

44 45 53 54 threatProcessing.js

file://file-3M4WXp7e7EpCxjsGFAB3Rj

55 56 mitreAttackMapping.js

file://file-1TTK9gqdknn85tDdBYJv5M

74 75 76 77 Atomic Design Methodology | Atomic Design by Brad Frost

https://atomicdesign.bradfrost.com/chapter-2/