ChatGPT

Enhanced Requirements Dashboard – Technical Architecture Design

Introduction

The Enhanced Requirements Dashboard for the Cyber Trust Sensor Platform will be built with a **modular, scalable React architecture**. We adopt an **atomic design** component structure and a feature-based modular approach to manage complexity. The goal is to create a secure, maintainable system that can integrate real-time data and external frameworks in the future. Key architectural priorities include:
- **Reusability and Maintainability:** Breaking the UI into small, reusable components to reduce development friction and ease onboarding [1].
- **Consistency:** Enforcing a unified design system for a coherent user experience across the app [1].
- **Scalability:** Organizing code to support new features and pages (e.g. adding the resource swimlane view) without monolithic growth. This includes optimizing performance via code-splitting and lazy loading of feature modules.
- **Secure by Design:** Following best practices (least privilege, input validation, dependency auditing) throughout development, and isolating components to limit side effects. The design will make it easy to integrate authentication/authorization checks and to handle sensitive data safely.

Architecture Overview

Overall Structure: The application is structured as a collection of feature modules within a React single-page app. Each feature (e.g. Requirements, Capabilities, Resource Planning, Threat Intelligence, etc.) encapsulates its own components, state logic, and services. This follows a modular design pattern where domain-specific functionality is grouped together for clarity and separation of concerns. For example, a typical project might have an `auth` module containing its own components, hooks, and services [2]. In our case, we will have modules for **Capabilities**, **Requirements**, **Resource Planning**, etc., each managing its own part of the state and UI.

Atomic Design Adoption: We apply Atomic Design principles to organize our React components into a hierarchy of **atoms, molecules, organisms, templates, and pages** [3] [4]. This provides a clear mental model and folder structure for the UI components [5] while aligning with React's component-based philosophy. By using Atomic Design, we promote code reuse and consistency and make the UI easier to extend or modify [1]. The **folder structure** will reflect this hierarchy (e.g. `src/components/atoms`, `src/components/molecules`, etc.) [5]. Major benefits of this structured approach include easier maintenance, improved collaboration among the team, and optimized performance through deliberate component composition [1].

Feature Modules & Shared Utilities: Each feature module may define its own higher-level components and may leverage shared lower-level components from the atomic library. Shared utilities (such as data formatting functions or hooks for common filters/analytics) reside in a utilities layer. We maintain a **service layer** (discussed later) to abstract data access, which makes it easy to switch from mock data to real APIs without changing UI components. Overall, components consume data via context hooks and

services, rather than hard-coding any data access, ensuring a clean separation between presentation and data logic.

Component Structure (Atomic Design)

Figure: The five levels of Atomic Design (Atoms, Molecules, Organisms, Templates, Pages) that form the basis of our component architecture [6].

Our component library is organized using **Atomic Design**, breaking the interface into hierarchical layers of components [3] [4]:

- **Atoms:** Fundamental UI elements that cannot be broken down further. These include basic tags and controls such as buttons, icons, input fields, labels, and small display elements (e.g. a text badge or an icon). Atoms are **stateless**, presentational components – for example, a `<Button>` or an `<Icon>` component. They define the base styling and can receive props but do not manage any complex logic on their own [7].

- **Molecules:** Groups of atoms combined into a small functional unit. A molecule is a relatively simple composite component – for example, a form input with an associated label and button can be a **SearchBar** molecule composed of an input (atom), a label (atom), and a button (atom) [8] [9]. Molecules encapsulate a specific function or piece of UI (like an input field with validation icon, or a card that combines an icon and text). They may handle basic user interactions in coordination with their child atoms, but typically **do not hold state** beyond perhaps local UI state.

- **Organisms:** Larger sections of the interface, composed of multiple molecules and/or atoms working together. Organisms form distinct, complex components of the app – for example, a **Capability Card** that displays a capability's name, description, and a progress bar (built from smaller components), or a **Header** organism that contains the logo, navigation menu, and search bar [10] [11]. Organisms can contain local state or handle more complex logic than molecules. In our dashboard, examples of organisms would include the **Requirements Table** (a table listing requirements, composed of table row molecules), the **StatCard** (a card showing a statistic with icon and value), or the upcoming **ResourceSwimlane** component. Organisms often correspond to meaningful sections of a page (e.g. a sidebar, a toolbar, a list widget) and can be reused across pages.

- **Templates:** Page-level layout components that arrange atoms, molecules, and organisms into a specific structure without being tied to real data. A template defines the **layout and placeholders** for content – for example, a dashboard layout template might specify a sidebar area for navigation (filled by a Navigation organism), a header area (with a Header organism), and a main content area for the active view. Templates typically combine multiple organisms to form a full page structure [12]. In our application, we might have a **DashboardTemplate** that includes the common layout (sidebar + header + content region) used by all dashboard pages. Templates themselves may manage state for layout concerns (like toggling sidebar collapse, as already done via UI state), but they delegate feature-specific state to the organisms within.

- **Pages:** Concrete pages or views that inject real content/data into Templates to form a complete screen [4]. A Page is often tied to a route (e.g. `/capabilities` page) or a view mode in the app. In our case, since the dashboard is a single-page application that switches views, each view (Capabilities view, Requirements view, Resource Planning view, etc.) can be thought of as a

"Page" component. Pages assemble the appropriate organisms (or use a Template) and pass in actual data from context or services. For example, the **RequirementsPage** would use the DashboardTemplate and populate the content area with the Requirements Table organism, filtering and listing actual requirements data. Pages typically *do not define new layout structure* – they reuse Templates – but they orchestrate which organisms appear and connect them to state or context.

This atomic decomposition enhances reusability and consistency. A small change to an atom (say, a Button style) propagates consistently across all molecules and organisms that use it, enforcing a cohesive UI. It also aligns with the principle that **smaller components are stateless**, while **stateful logic is lifted to higher-level components** (organisms or templates) [13]. For example, an atom or molecule will not typically contain context logic or data fetching; those concerns live in organisms/pages which then render atoms/molecules as children. This separation makes components easier to test and reuse, and it avoids duplicating complex logic.

The project's **file structure** will mirror this hierarchy for clarity [5]. For instance:

```
src/
 ├── components/
 │   ├── atoms/      # e.g. Button.jsx, Icon.jsx, Input.jsx, Label.jsx,
etc.
 │   ├── molecules/  # e.g. SearchBar.jsx, StatCard.jsx, FormField.jsx
 │   ├── organisms/  # e.g. Header.jsx, CapabilityCard.jsx,
RequirementsTable.jsx
 │   ├── templates/  # e.g. DashboardLayout.jsx, WizardLayout.jsx (if
needed)
 │   └── pages/      # e.g. RequirementsPage.jsx, CapabilitiesPage.jsx,
ResourcePlanningPage.jsx
 ├── context/        # Context providers and hooks for state (by feature)
 ├── services/       # Data services (API interactions or mock data
adapters)
 ├── hooks/          # Reusable hooks (e.g. useFilteredRequirements,
useAnalytics)
 ├── utils/          # Utility functions (e.g. CSV export, calculations)
 └── ...
```

This organization ensures that **presentational components** (atoms, molecules) are kept separate from **page-specific components**, and that related files are easy to locate. Developers can navigate the codebase quickly (e.g., all basic form controls are in `atoms/`, all composite widgets in `organisms/`, etc.), improving team collaboration and code quality [1]. It also sets the stage for a shared component library (we could integrate Storybook for documentation of atoms/molecules, as suggested in atomic design methodology [14]).

State Management and Contexts

State Scope and Separation: In line with React best practices, the design handles state at the appropriate component levels. We avoid cluttering low-level components with global state. **Atoms and most molecules are stateless UI components**, while state is managed by higher-level components like organisms or pages [13]. For example, an atom like `<TextInput>` simply displays a value and

triggers an `onChange` callback; it does not know where the data comes from. An organism like `<RequirementsTable>` or `<ResourceSwimlane>` may hold local UI state (e.g. current sorting or an open/closed section) and will consume global state via context for data. Templates or page components may manage broader UI state (like the current selected view or modal visibility), as we see in the existing dashboard reducer for view modes. This division makes each component simpler and more focused.

Contexts and Stores: We leverage React Context API to manage shared state across components in a robust, modular way. Rather than a single monolithic store, we will use **multiple scoped context providers** to isolate state by feature/domain. This means, for example, we will have:

- a **RequirementsContext** providing the list of requirements and operations on them (create, update, delete),
- a **CapabilitiesContext** providing capability data and operations,
- possibly a **ResourcePlanning/TeamContext** for resource (people) data and allocations, and so on.

Each context is exposed via a custom hook (e.g. `useRequirementsData()`, `useCapabilitiesData()`) to conveniently retrieve state and actions for that domain. This pattern is already present in the current system (with hooks like `useRequirementsData()` and `useCapabilitiesData()` providing data and mutation functions). We will extend and formalize it for new features.

Using **multiple contexts ensures separation of concerns and minimizes unnecessary re-renders** – components only subscribe to the state they care about [15]. For instance, the requirements table can re-render when requirements data changes without causing the entire app or unrelated parts (like threat intelligence views) to re-render. Likewise, the capability context can manage capability status and metadata independently. As Kent C. Dodds notes, context need not be global to the whole app; it's often better to have logically separate contexts for different data [15]. Our design follows this advice by **scoping providers to the relevant part of the component tree**. For example, we might wrap the Dashboard area in providers for Requirements, Capabilities, and other GRC data, while an authentication context might wrap the whole app (to provide user info globally). The new **Resource Planning** features could be wrapped in its own provider if it introduces distinct state (e.g. allocation of people), or it might reuse existing providers (if it mainly uses requirements and capabilities data).

Context Relationships: The contexts are largely independent but will sometimes intersect. For example, a "capability" has many "requirements". In the UI, when viewing a specific capability, we filter requirements by a capabilityId – this is currently handled by the dashboard component using the two datasets. In the new architecture, the **RequirementsContext** might still hold the master list of requirements, and the CapabilitiesContext holds the list of capabilities. If a capability needs to display its associated requirements, it can either:
- call a selector function or hook (e.g. `useRequirementsData().getRequirementsByCapability(capId)`), or
- rely on a derived state provided by the RequirementsContext (we could provide a computed map of capability→requirements if performance demands).

This approach ensures there is a **single source of truth** for requirements data (the RequirementsContext store). We avoid duplicating state across contexts. If a requirement is updated (e.g. marked completed) via `useRequirementsData().updateRequirement`, the components that use that context (including capability views and resource views) will see the change immediately. This consistency is crucial if **a requirement can belong to multiple capabilities** or workstreams – we do

not want separate copies diverging. Instead, a requirement could carry references to related capabilities, and each context subscriber can react accordingly.

Shared State Patterns: To manage **shared or cross-cutting state**, we will introduce either **utility hooks** or **combined context providers** where appropriate. For example, the dashboard's UI state (current view mode, filters, modals) could be managed by a Dashboard UI context or reducer. This UI state context can coordinate between features (so that, for instance, selecting a capability in the UI triggers the Requirements view to show that capability's requirements). The current implementation uses a reducer (`dashboardReducer`) in the RequirementsDashboard component for UI state; we can refactor this into a context provider (e.g. `<DashboardUIProvider>`) that wraps the dashboard pages. This preserves the state (and keyboard shortcuts, modals, etc.) across the entire dashboard and allows any component to dispatch UI actions via context rather than prop drilling the dispatcher.

Custom Hooks and Selectors: We will continue using custom hooks for derived data. The existing `useFilteredRequirements()` and `useAnalytics()` are good examples of separating computation from rendering. We'll maintain this pattern: any expensive or reusable computation on context state (such as filtering, aggregating maturity scores, calculating totals) will live in a hook or utility function, not directly in the component's render. For example, we might provide a hook like `useCapabilitySummary(capabilityId)` that uses both RequirementsContext and CapabilitiesContext to compute that capability's completion percentage, aggregated effort, etc., and returns a memoized result. This keeps our components lean and declarative; they just consume these hooks to get data. In terms of implementation, these hooks may internally use `useContext` from multiple providers or use `useMemo` to optimize calculations.

For global read-heavy state (like lists of requirements), if performance becomes a concern, we could consider optimizing context usage (React context updates can be optimized by splitting value or using selectors). Currently, the data volumes seem moderate (capabilities and requirements lists), but our design is mindful of scalability – we can partition context or use libraries like Zustand or Redux Toolkit in the future if the state grows significantly. At this stage, the context + hook approach provides simplicity and adequate performance.

In summary, **state is managed in feature-specific contexts** with well-defined interfaces (via hooks). Components access and update state through these hooks. This results in a clear separation: UI components focus on rendering, while context providers and services handle data management. This modular state management will make our app easier to extend (for example, adding a new context for a new feature is straightforward) and to integrate with external data sources down the line.

Resource Swimlane Module (Team Planning View)

One of the major enhancements is the introduction of a **Resource Swimlane** view – a new organism (and page) focused on team planning. This component will provide a visual allocation of **people (resources)** to **capability and requirement workstreams** and track each work item's **maturity, effort, and duration** status. We will design the Resource Swimlane as a highly modular component so it can be reused or extended in the future (for example, to drill down into individual resources or to integrate with scheduling tools).

Component Structure: The Resource Swimlane can be considered an **organism** (or even a small ecosystem of organisms) composed of smaller building blocks:
- *Resource Lane:* an element representing a single resource (e.g. a person or team). This could be a row in a list or a swimlane in a Kanban/timeline view. It typically includes the resource's name (and possibly

avatar or role) and containers for that resource's assigned items.
- *Assigned Item Card:* a compact representation of a requirement (or capability) that the person is working on. This could be a card or bar that shows key info like the requirement title, its capability or category, current maturity level, effort estimate, and duration or due date. For example, a card might display "Requirement X – Maturity 3/5 – 5d effort – due Nov 10". If we implement it as a timeline, this might be a bar spanning the duration on a calendar axis, placed in the row corresponding to the resource. In a simpler list view, it might just be a list of such items under each resource.
- *Maturity Indicator:* a UI element (likely reused, possibly the existing **MaturityIndicator** component) to visually show the maturity level of a requirement (e.g. colored dots or a gauge icon).
- *Effort/Duration UI:* perhaps a small progress bar or icon representing effort or a time span. We might use a clock icon with hours, or a bar whose length corresponds to duration. These are mostly atoms/molecules that we can create or reuse (for instance, the existing StatCard or progress bar styles can be adapted).

We will likely create the Resource Swimlane as a **feature module** under `components/organisms` (and possibly a page in `components/pages`). Internally, it might be broken down further: for example, `<ResourceSwimlane>` could map over a list of resources and render a `<ResourceLane>` for each, which in turn maps over that resource's tasks and renders an `<AssignedTaskCard>` for each. Breaking it down this way keeps each piece manageable. The styling will use a responsive flexbox or grid to create "swimlanes". On large screens, we could present a multi-column layout (e.g. lanes side by side or a horizontal timeline); on smaller screens, it could collapse to a stacked list per resource.

Data Model for Resources: To support this view, we introduce the concept of a **Resource** in our state. A Resource likely represents a team member. We will maintain a list of resources (team members) – potentially via a new context (e.g. `TeamContext`). This context would store all people available to assign, along with their attributes (name, role, capacity, etc.). If the platform integrates with an identity system or HR feed, this context would be the consumer of that data. Initially, we can define a static list of resources (or derive it from the assignments present in requirements). In addition, each Requirement should have an attribute for the assigned resource(s). The current data model doesn't include assignees, so as part of this enhancement we will extend the **Requirement** object to include an `assignedTo` field (likely a user ID or list of user IDs). This allows a single requirement to be assigned to one or multiple people if needed (though typically one owner per task). The **ResourceSwimlane** will use this relationship to group requirements by their assigned resource.

Connecting to Capability & Requirement State: The ResourceSwimlane is inherently a cross-section of **Capabilities** and **Requirements** data from the perspective of **Resources**. It will interact with existing contexts as follows:
- It will consume the **RequirementsContext** to get all requirements (or perhaps a filtered subset if we only show active/in-progress ones in the planning view). From each requirement, it will extract fields like `assignedTo`, `maturityLevel`, `effortEstimate`, `duration` (we may need to add effort and duration fields if not present; currently we have `costEstimate` which could serve as effort/cost, and perhaps we interpret `status` to infer duration or due dates might be added).
- It will consume the **CapabilitiesContext** mainly for reference (e.g., to display the capability name or category a requirement belongs to, or to filter by capability). For instance, the item cards could show a tag or color indicating the capability. We might color-code requirements by their capability (the system already color-codes capability cards). This helps users see which domain each task belongs to.
- It will use a **Team/Resource Context** (if implemented) to get the list of resources and their details (like full name, avatar, and possibly workload capacity if we show load). If we don't have a dedicated TeamContext, the resources could be derived from requirements (collect all unique `assignedTo` values) – but a context is cleaner and allows adding unassigned team members for planning purposes.

Behavior and Interaction: The Resource Swimlane will allow users (most likely managers or team leads) to **plan and track work**. Key interactions we plan to support:
- **View allocation:** The swimlane clearly shows which requirements each person is responsible for. This gives a quick overview of distribution of work. If implemented as a timeline, one can see if timeframes overlap or if someone is overbooked. If as a simple grouped list, one can still see count of tasks per person.
- **Update assignments:** Ideally, the UI will allow reassigning a requirement to a different person (e.g. via drag-and-drop of a card to another resource's lane, or an edit action on the card). The component will handle such events by updating the requirement's `assignedTo` field (calling an action from RequirementsContext, like `updateRequirement`). This demonstrates the bidirectional connection: the swimlane reads from context and also dispatches updates to it. With our context-based state, moving a task to a different person triggers a context update, which will cause the ResourceSwimlane to rerender reflecting the change (as well as any other view showing that requirement, e.g. the Requirements table).
- **Filter/Scope:** We might provide filtering controls, for example to filter the swimlane to a particular capability or status. This could reuse existing filtering logic (the Dashboard UI context has filters). E.g., the user could filter to show only requirements of Capability X – then each resource lane would only list tasks from that capability. Or filter by status to only see "In Progress" work. These filters can be managed via the same filter state in the dashboard context (so that filter state is consistent across views). The swimlane component will need to respect those filters (similar to how the Requirements table uses `useFilteredRequirements`).
- **Maturity and Effort tracking:** Each assigned task card will display the maturity level (e.g. "Level 2 – Defined" or a visual indicator). If maturity is something that evolves over time (perhaps updated as the requirement implementation progresses), the swimlane provides a place to update or at least monitor it. We might allow inline editing of maturity (if a user has completed a control and raises its maturity score, they could adjust it here). If so, that would again call an update on the requirement via context. Effort and duration could be purely informational (estimated effort, planned duration) or interactive (maybe adjusting duration if timeline-based). Initially, we can show the values and later consider making them editable or integrated with scheduling logic.

Aggregating Data: The swimlane view also provides an opportunity to **aggregate metrics** by resource or by capability. For example, we could show a summary per resource (like total number of tasks, total effort sum, average maturity of their tasks). This can help project managers see who has the most work or where maturity is lagging. These aggregations can be computed via selectors: e.g., `totalEffort(resourceId) = sum of effort of all requirements assigned to that resource`. We can implement such calculations either within the component or via a custom hook (`useResourceStats(resourceId)`). Similarly, we might aggregate by capability across resources: for instance, an overall view that Capability X requires 50 person-days of effort (sum of all tasks under it) and is 60% complete. These cross-cuts relate to **effort/maturity aggregation**. We plan to handle this via the analytics utilities (there is already a `useAnalytics(requirements)` hook – it could be extended to compute stats per capability or per resource). Managing shared state for these calculations will be important if requirements belong to multiple capabilities – we will count them appropriately in each context but ensure the source data is one. Using **normalized data modeling** (each requirement has one record, with references) prevents double-counting issues: for example, if a requirement is linked to two capabilities, it should ideally count toward both capabilities' effort, but the requirement's data (like its effort estimate) is stored in one place. By referencing it from both capability views, we ensure consistency. We may implement a structure where a requirement can have an array of `capabilityIds` if multi-mapping is needed, and then adjust filter/aggregation logic to include it in multiple places. This approach keeps shared state accurate without duplication.

Integration into the Dashboard: The ResourceSwimlane will be integrated as a new **page/view** within the Enhanced Dashboard. We will add a new navigation entry (e.g., "Team Planning" or "Resources") in

the sidebar menu alongside Capabilities, Requirements, etc. Selecting this will switch the dashboard to the Resource Planning view. If using react-router, this would correspond to a new route (for example, `/dashboard/resources`). If continuing the current viewMode state pattern, it would correspond to a new `viewMode: 'resources'` state and be handled similar to existing ones. In either case, the ResourceSwimlane component will be loaded and displayed in the main content area when activated. We will also ensure this component is **lazy-loaded** (more on routing/lazy loading below) to avoid increasing the initial bundle size. The ResourceSwimlane will utilize the contexts as discussed (hence it should be rendered inside the providers for requirements, capabilities, etc., which is naturally the case if it's within the Dashboard layout).

By treating the ResourceSwimlane as a first-class module, we ensure that team planning becomes an integral yet isolated part of the platform. The architecture will allow the **ResourceSwimlane to evolve** (e.g., adding more sophisticated scheduling, linking to calendars or external project management systems) without affecting unrelated parts of the app, thanks to the modular separation. At the same time, because it reuses the centralized contexts for data, it stays in sync with the rest of the dashboard (all views are looking at the same underlying requirements and capabilities data).

Routing, Navigation & Lazy Loading

Routing Structure: To improve navigability and scalability, we will use React Router (v6+) to define distinct routes for major sections of the dashboard. Each "page" or view (Capabilities, Requirements, Resource Planning, etc.) gets a route path. For example:
- `/dashboard/overview` – Overview page
- `/dashboard/capabilities` – Capabilities page
- `/dashboard/requirements` – Requirements page
- `/dashboard/resources` – Resource Planning (new)
- (and similarly for Threat Intelligence, Analytics, etc.)

Using explicit routes has several benefits over solely using internal state toggling: it allows deep linking (a user can bookmark the Resource Planning page URL), browser navigation (back/forward), and easier code-splitting per route. In the current system, view switching is handled by internal state (`viewMode` in a reducer). We can still maintain a similar state for quick switching, but we will synchronize it with the router or transition to using the router's state. For backward compatibility or simplicity, we might start by keeping the internal `viewMode` but gradually refactor to actual routes. A hybrid approach could be to use router but have a single component manage the switching (mapping route param to viewMode). Ultimately, adopting routing is advisable for a "world-class" architecture due to clarity and URL addressability.

Navigation: The sidebar menu will be updated to use React Router's navigation mechanism (e.g. using `<Link>` components) instead of simply dispatching a state change. Each menu item will correspond to a route (as listed above). For example, clicking "Capabilities" would navigate to `/dashboard/capabilities`. We will ensure that the active route is highlighted in the menu (React Router's hooks or NavLink can help with that). This change decouples navigation from the dashboard's internal state, making the app behavior more transparent. (If for some reason we do not use Router, we will keep the dispatch but the outcome is similar – in either case, the ResourceSwimlane gets integrated with a menu entry.)

Page Layout Template: We will create a **DashboardLayout template** that contains the persistent parts of the dashboard: sidebar, header, and possibly any common context providers. Each route will render

this template and inject the specific page content. For example, in code, our routes might be structured as:

```
<DashboardLayout>
   <Routes>
      <Route path="capabilities" element={<CapabilitiesPage/>}/>
      <Route path="requirements" element={<RequirementsPage/>}/>
      <Route path="resources" element={<ResourcePlanningPage/>}/>
      ...
   </Routes>
</DashboardLayout>
```

Here, `DashboardLayout` is a template organism that includes the sidebar and header. The individual Page components (CapabilitiesPage, etc.) correspond to what was previously toggled via viewMode. They will render the relevant organisms (e.g. CapabilitiesPage might render a list of Capability cards; ResourcePlanningPage renders the ResourceSwimlane organism). This approach ensures **code separation per page**, and combined with lazy loading, means we only load the code for the page the user is viewing.

Lazy Loading: To keep initial load times fast, we will implement **lazy loading** for heavy components and feature modules. React's `lazy()` and `Suspense` will be used to split the bundle. Each page component can be dynamically imported. For instance, `const ResourcePlanningPage = React.lazy(() => import('./components/pages/ResourcePlanningPage'));`. The router can then reference this lazy component with a fallback loader. This way, if a user never opens the Resource Planning section during a session, its code (and any related library code like charts) won't be downloaded. Given that some parts of our app (e.g. the MITRE Attack Navigator, charts for analytics) might include large third-party libraries (like Recharts, D3, etc.), lazy loading those routes will significantly improve performance. We have to ensure any context providers needed are at a level above the Suspense boundary so that data is available when the chunk loads.

In cases where internal state switching is retained (for quick toggling of sub-views without route changes), we can still lazy-load the content for rarely used views. For example, if "Threat Intelligence" or "Standards" view is not frequently used, we could wrap their component rendering in a `<Suspense>` with dynamic import. The current code already imports some components normally (MitreAttackNavigator, etc.). We will refactor those to load lazily if they are large. The *Enhanced Requirements Dashboard* will especially lazy-load the **ResourceSwimlane** module, since it's new and likely not needed on initial landing page.

Routing Guards and Secure Access: With the introduction of routing, we can also integrate route-based permission checks. For instance, if certain roles should not access the Resource Planning page, we can enforce that via a route guard (checking the AuthContext for user role, and redirecting if unauthorized). This ties into the secure-by-design approach, ensuring that even on the front-end we don't inadvertently expose links or data to unauthorized users (the backend should enforce truly, but the front-end can provide usability by hiding disabled sections).

View/Page Layout Approach: We will maintain a responsive layout. The DashboardLayout will likely continue using a sidebar that collapses on mobile (as current logic does). Each page's content should be designed to reflow or provide horizontal scrolling if needed (for example, the swimlane timeline might scroll horizontally on smaller screens). We'll leverage CSS grid/flex and existing tailwind styles for this.

Because each page is separate, we can tailor the layout: e.g., the ResourcePlanningPage might use a horizontal scroll for the timeline portion while the CapabilitiesPage might use a grid for capability cards. The template ensures consistent header/sidebar across all.

In summary, adopting a route-centric approach with lazy loading improves the **scalability and performance** of the app. As the platform grows (more frameworks, more views), this architecture can handle it without a tangled state machine. The navigation is clear and the code for each section is isolated and only loaded when needed. This is crucial as we integrate more real-time data and possibly external iframes or widgets – those can be compartmentalized into their own routes or code-split chunks.

Data Services and API Integration

Although the current system uses mock data, we are architecting with an eye toward **future API integration**. This means establishing clear boundaries between the UI and data sources. We will introduce a **data service layer** (under `src/services/` or similar) to handle all data fetching, transformations, and updates. Each feature or data type can have a dedicated service module (for example, `requirementsService.js`, `capabilitiesService.js`). These service modules will expose functions like `fetchRequirements`, `addRequirement`, `updateRequirement` etc. In the current mock setup, these might simply read from local storage or a static JSON and update in memory. In the future, these functions will make HTTP requests to an API endpoint (e.g. using `fetch` or a library like Axios or React Query). By calling these through the context hooks, we encapsulate the data access logic. For instance, the `useRequirementsData` hook will internally call `requirementsService.fetchAll()` on initial load, and use `requirementsService.update(req)` when an update is triggered. The rest of the app doesn't need to know if data comes from a local mock or a remote server – that can be toggled via configuration.

API Design Considerations: We anticipate endpoints such as: - `GET /api/requirements` (with query options for filtering)
- `POST /api/requirements` (to add new requirement)
- `PUT /api/requirements/:id` (to update a requirement's fields, including status, maturity, etc.)
- `GET /api/capabilities` (and similar POST/PUT if capabilities can be added or updated by users)
- `GET /api/team` or `GET /api/users` (to list team resources for assignment)

Our service layer will be designed to handle responses from these endpoints and map them into our context state. We will likely maintain TypeScript interfaces or PropTypes for the data models (Requirement, Capability, etc.) so that conversion from API JSON to front-end model is consistent and type-safe. Even if we remain in JS for now, having a defined shape in one place (like a normalizer) helps.

Mock Data Strategy: During development and until the API is ready, we can implement the service functions to use mock data. For example, `fetchRequirements` might call a utility like `generateMockData()` (as currently done) [16] and populate context state. We'll structure the code such that switching to real API is as simple as changing the implementation of that service function. This could be facilitated by environment configuration – e.g., a flag `USE_MOCK_DATA=true` that picks a different set of service implementations. We could even use dependency injection or a simple conditional inside the hook (`if (mock) generateMockData() else await axios.get('/api/requirements')`). The goal is **no changes needed in components or context usage** when the data source changes – they call the same hook methods.

Real-time Data and External Integration: The architecture anticipates real-time updates (for instance, if multiple users collaborate or if data from external systems flows in). To handle real-time, we could incorporate WebSocket or Server-Sent Events connectivity in the future. Our context providers are a good place to integrate that: e.g., a RequirementsContext could open a WebSocket upon mounting that listens for any requirement updates broadcast by the server, and then dispatches them to update state. The components would update reactively. Because our design concentrates update logic in one place (the context or reducer), adding such real-time behavior is straightforward. We maintain the invariant that UI components never directly mutate data; they always go through context actions. So if an external event comes in, it also goes through the same action pipeline, preserving consistency.

For external frameworks or integrations (like perhaps pulling data from a project management tool or GRC database), we again rely on the service abstraction. We might create integration services (e.g. an adapter that fetches data from a JIRA API or from a CSV import). The UI remains the same, but we add a service method and perhaps call it in a context action.

Security (API Considerations): When we move to real APIs, we will ensure to follow security best practices: use HTTPS for all calls, include proper authentication tokens (possibly the AuthContext provides JWT or API keys to the service layer). We will handle sensitive data carefully – for example, if any personal data of team members is displayed, ensure compliance with privacy (show only what's needed and secure it). Also, to protect against XSS or injection, any data coming from the API will be sanitized or safely rendered (React by default escapes content, which is good). We will avoid using `dangerouslySetInnerHTML` unless absolutely necessary (and even then, only with trusted content).

Error Handling and Loading States: The service layer and contexts will also manage loading and error states for API calls. For instance, `useRequirementsData()` might expose a `loading` boolean and an `error` object (we see this pattern already: `loading, error` are returned [17]). We will propagate those to the UI (e.g., show a spinner or an error message in the relevant component if data fails to load). This ensures a robust user experience even when dealing with latency or failures. For the ResourceSwimlane, if an API call is needed (say to fetch utilization data), it should show a loading indicator within that view until data is ready.

In summary, **API preparation** in our design means creating a clear separation via services and contexts such that switching from mock to real data is seamless. The architecture is scalable to real backend integration and even multiple data sources. This also contributes to security, as all data flow can be monitored and controlled at these choke points (we can, for example, add centralized input validation or logging in the service layer for anything coming from external sources).

Security & Scalability Considerations

Security is baked into the design from the start ("secure-by-design"). Even though much of security enforcement happens on the backend, the front-end architecture plays a role in ensuring data is handled safely and the attack surface is minimized:

- **Access Control in UI:** We will use the Auth context (or JWT tokens) to conditionally render UI elements based on user roles/permissions. Sensitive actions (like deleting a requirement or viewing certain data) will only be shown if the user's role allows it. This is a usability and security measure to prevent unauthorized actions. (Backend will still validate, but hiding it on UI reduces temptation and accidental attempts.)

- **Isolated Components:** By keeping components self-contained (as atoms/molecules) and not relying on global mutable variables, we reduce the chance of unintended interactions or data leaks between parts of the app. Each context provider acts as a boundary. This also helps in security reviews – we can audit each context's methods for safety. For example, if a context method modifies data, we ensure it cannot be called without proper user action, and perhaps in the future tie it to permission (e.g. only an admin context could expose a deleteCapability method).

- **Sanitization and Validation:** Input components (atoms like `<TextInput>`) will be complemented by validation logic at the form or organism level. We'll validate inputs (e.g. no scripts in text fields, proper formats for numbers/dates) before sending to the backend. This is part of secure-by-design to catch issues early. Also, any data coming from APIs will be handled carefully – while React escaping means XSS is less of an issue, we still avoid inserting any unchecked HTML. If we need to display rich text from users, we will use safe libraries or sanitize it.

- **Dependency Security:** We will keep an eye on third-party libraries (like UI frameworks or chart libraries) and ensure they are up-to-date to patch vulnerabilities. As part of the project plan, a security review task will include scanning dependencies (using tools like `npm audit`) and addressing any flagged issues.

- **Performance & Scalability:** Scalability also refers to handling growth in data volume and users. The chosen architecture (with context separation and lazy loading) will help performance as the app grows. By lazy-loading, we keep initial load fast and avoid overwhelming the browser with unused code. By splitting contexts, we avoid massive re-renders and can optimize each part independently. If the number of requirements or capabilities grows into the thousands, we can introduce virtualization in the table or pagination; our separation of concerns makes that easier (we could swap the RequirementsTable organism implementation for a virtualized list without affecting other components).

- **Concurrency and Consistency:** In a scenario where multiple users edit the data simultaneously (future real-time collaboration), our single-source-of-truth contexts combined with real-time updates will maintain a consistent view. We might add optimistic update patterns in the service layer to give snappy UI feedback while awaiting server confirmation. The design is amenable to such enhancements.

- **External Framework Integration:** If we integrate external widgets or iframes (for example, an external GRC visualization), we will sandbox them appropriately. For instance, if embedding an external site via iframe, use the `sandbox` attribute and restrict origins. But a more likely scenario is using external libraries (like a graph visualization library). We will incorporate them in a way that does not compromise the app (e.g., not eval-ing code, using well-maintained libraries). Because our architecture isolates features, adding a new library for one view (like a specialized compliance chart) will not affect the rest of the system – it can be lazy-loaded with that view and kept encapsulated.

- **Testing & QA:** As part of being secure and robust, we plan comprehensive testing. Unit tests will cover critical context logic (ensuring that reducers only allow valid state transitions, etc.). We'll also write tests for components to ensure they properly reflect state (for example, the ResourceSwimlane should correctly display tasks per person and handle an assignment change). Automated tests help catch regressions that could introduce vulnerabilities or break security

assumptions (like accidentally letting an unauthorized user access something due to a logic bug).

The architecture is designed to be **scalable** in terms of both features and team size. With a clear structure, multiple developers can work concurrently on different modules (one on ResourceSwimlane, another on Capabilities) with minimal overlap. This parallelizability accelerates development – which is reflected in the project plan tasks breakdown.

In conclusion, this technical architecture balances **modularity, clarity, and preparedness for the future**. By adhering to React best practices (atomic design, context-driven state management, lazy loading) and by planning for secure and scalable growth, the Enhanced Requirements Dashboard will be robust foundation for the Cyber Trust Sensor Platform's ongoing development. The following project plan outlines the implementation steps to realize this design.

Project Plan (CSV)

Below is a CSV-formatted project plan that breaks down the redesign and development into clear tasks with priorities, dependencies, and assigned roles. Each task is scoped to be estimable and highlights any prerequisites and responsibility.

```
Task,Description,Priority,Dependencies,Owner
Architecture Setup,Establish atomic design structure and base project layout
(folder setup and initial components),High,,Tech Lead
Context Design,Design application state management with separate contexts for
capabilities & requirements (and others as needed),High,Architecture
Setup,Tech Lead
UI Library Creation,Build base UI components (atoms & molecules) and
establish style guide,High,Architecture Setup,Frontend Developer
Resource Module Design,Design Resource Swimlane component structure and sub-
components,High,Architecture Setup; Context Design,Frontend Developer
Resource Context Integration,Implement state management for resource
assignments and link to capability/requirement data,Medium,Context
Design,Frontend Developer
Feature Wiring,Integrate ResourceSwimlane into dashboard UI: add route and
navigation entry; wrap component with required context
providers,High,Resource Module Design; Resource Context Integration,Frontend
Developer
Routing & Lazy Loading,Configure routing and lazy loading for new and
existing pages,Medium,Architecture Setup,Frontend Developer
API Service Layer Prep,Create data service interfaces for future API
(abstract mock data calls),Medium,Architecture Setup,Full-stack Developer
Testing & QA,Write unit and integration tests for contexts and components;
perform UI testing,High,Feature Wiring,QA Engineer
Security Review,Perform security assessment (dependency audit & code review)
and address findings,High,Feature Wiring,Security Engineer
Documentation,Document architecture and provide usage guidelines for
team,Medium,Feature Wiring,Tech Lead
```

---

[1] [3] [4] [7] [12] [14] Atomic Design Pattern: Structuring Your React Application | by Roopal Jasnani | Medium
https://rjroopal.medium.com/atomic-design-pattern-structuring-your-react-application-970dd57520f8

[2] Scalable and Modular React Native Expo Folder Structure 2025 | by Md Alishan Ali | Medium
https://medium.com/@md.alishanali/scalable-and-modular-react-native-expo-folder-structure-2025-606abc0bf7d6

[5] [8] [9] [10] [11] [13] The Future of UI Development: React Atomic Design Explained
https://www.dhiwise.com/post/the-ultimate-guide-to-react-atomic-design-principles

[6] Atomic Design Methodology | Atomic Design by Brad Frost
https://atomicdesign.bradfrost.com/chapter-2/

[15] How to use React Context effectively
https://kentcdodds.com/blog/how-to-use-react-context-effectively

[16] [17] RequirementsDashboard.jsx
file://file-72UbX3Nko8QYkqU3JcghdX