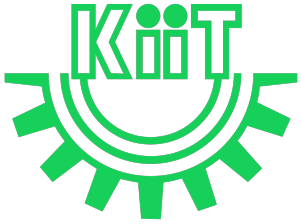


KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)

Deemed to be University U/S 3 of UGC Act, 1956

Project Report

Topic: Email Threat Monitoring

Github Repository: https://github.com/mayankagarwal-01/threat-map

By,

Name: Mayank Agarwal
University Roll No.: 23051999
University Email: 23051999@kiit.ac.in

Table of Contents

Introduction

This report provides an exhaustive technical analysis of the Threat Map web application, a modular, cloud-based platform engineered for the monitoring, analysis, and management of email-based threats within organisational settings. Initiated as an internship project, this endeavour embodies both the stringent professional cybersecurity engineering standards and the distinctive learning and innovation prospects afforded by an academic-industry partnership.

The development process encompassed comprehensive phases, including requirement gathering, system design, implementation, and rigorous testing. The architecture of the Threat Map web application incorporates advanced encryption protocols, real-time threat intelligence integration, and robust data privacy measures to ensure optimal security performance. Special emphasis was placed on scalability and adaptability, allowing the application to evolve with emerging cyber threats and organisational needs.

Key features of the application include dynamic threat visualisation, automated incident response mechanisms, and an intuitive user interface designed to enhance operational efficiency. The report also delves into potential areas for future enhancement, such as incorporating machine learning algorithms for predictive threat analysis and expanding support for multi-channel threat monitoring beyond email.

This document aims to serve as a comprehensive guide for cybersecurity professionals and stakeholders interested in understanding the technical intricacies, development methodologies, and strategic objectives that underpin the Threat Map web application.

Purpose and Scope

The primary objective of the Threat Map project is to deliver a customizable, extensible dashboard tailored to the detection and triage of suspicious email activity—a vector that remains among the most prevalent and damaging in contemporary cyber attacks. Recognizing the critical need for visibility into phishing, malware, and other email threats, the project scope encompasses: secure user authentication; ingestion and visualization of threat data; role-based administrative controls; file and remark management; and automated report generation.

The scope of this report is equally expansive. It details the project's design rationale, architectural decisions, technology choices, quality assurance procedures, and planned trajectories for future enhancements. By bridging hands-on technical depth with actionable insights, the document aims to serve both as an internal reference for developers and as a blueprint for stakeholders considering adoption or integration of Threat Map into their security operations.

Internship Context and Development Approach

Developed in the context of an internship, Threat Map was conceived as a real-world learning platform for applying best practices in secure web application design. The project was structured around **incremental, iterative development cycles**—each focusing on a distinct functional area, subjected to peer and supervisor review, and culminating in integrated feature sets. Guidance from experienced security engineers and IT professionals ensured technical accuracy, while regular demonstrations and feedback loops facilitated the early identification and remediation of bugs, usability issues, and feature gaps.

The internship context offered a dynamic interplay between guided exploration and independent problem-solving. Emphasis was placed on industry-relevant skills, such as implementing secure authentication via AWS Cognito, designing RESTful APIs backed by AWS Lambda, and leveraging serverless patterns for scalability and cost efficiency. This environment fostered a robust understanding of both cloud-native security architectures and the organizational realities of deploying threat monitoring systems under constraints of time, budget, and evolving requirements.

Why Email-Based Threat Monitoring?

Email continues to be a primary channel for the initial delivery of cyber threats, with phishing attacks consistently ranking among the leading causes of breaches and data loss. Consequently, organisations necessitate robust and adaptable tooling to illuminate the threat landscape within their email ecosystems, empowering analysts and administrators to accomplish the following:

- - Identify, triage, and monitor suspicious emails and campaigns
- - Maintain audit-compliant records of incidents and remediation actions
- - Coordinate responses across operational, IT, and cybersecurity teams
- - Extract actionable intelligence for ongoing defence improvement

By focussing on email threats, Threat Map addresses a tangible and high-impact concern for contemporary enterprises, effectively addressing the limitations often encountered with generic or prohibitively expensive commercial solutions.

Technology Stack and Stakeholder Alignment

Component	Description
JavaScript (Frontend)	Implements dynamic, single-page user interfaces, supporting secure, responsive analyst workflow
AWS Services (Backend)	Includes Cognito for authentication, Lambda for serverless compute, DynamoDB for NoSQL data storage, and S3 for report/file management

The design and feature set were selected with direct input from security stakeholders—analysts, system administrators, and security managers—who require rapid situational awareness, secure access controls, and the ability to generate actionable reports for compliance and management.

Report Structure

This document is structured into distinct technical modules, each providing a comprehensive analysis of a specific application component. These modules include detailed breakdowns, UI/UX design considerations, thorough testing and quality assurance analysis, system architecture diagrams, comparative analysis with industry tools, and forward-looking integration strategies. Each section is meticulously crafted to ensure clarity, transparency, and actionable guidance for both practitioners and decision-makers committed to the future of organisational email security.

Background: Threat Monitoring Systems and Use Cases

Modern cybersecurity is characterised by a dynamic and increasingly complex threat landscape. Organisations face persistent threats ranging from widespread phishing campaigns and commodity malware to sophisticated targeted attacks (Advanced Persistent Threats - APTs) orchestrated by state-sponsored actors or organised crime groups. In this environment, merely implementing preventive measures such as firewalls and antivirus software is insufficient. Effective defence requires continuous vigilance, the ability to detect malicious activity as it occurs, and swift, informed response. This is the fundamental role of threat monitoring systems.

Threat monitoring systems provide the crucial capability for organisations to gain visibility into their security posture and identify signs of compromise across their digital assets. These systems collect, aggregate, and analyse security-relevant data from diverse sources, including network devices, endpoints, applications, and logs. Their primary objective is to detect anomalous or malicious patterns of activity that indicate a potential or active threat.

Evolution of Threat Monitoring

The concept of threat monitoring has evolved significantly over time. Early approaches focused on basic network intrusion detection. Traditional **Intrusion Detection Systems (IDS)** and **Intrusion Prevention Systems (IPS)** primarily relied on signature matching to identify known malicious traffic patterns or exploits. While still relevant, this approach struggles against novel or rapidly evolving threats.

The complexity of modern attacks, which often involve multiple stages and vectors, led to the development of more integrated platforms. **Security Information and Event Management (SIEM)** systems emerged to aggregate logs and alerts from disparate security tools and IT systems into a centralized repository. SIEMs facilitate correlation analysis, allowing security analysts to connect seemingly unrelated events to identify larger attack campaigns. Simultaneously, **Endpoint Detection and Response (EDR)** solutions gained prominence, focusing on detailed monitoring of activities occurring on individual workstations and servers to detect post-compromise behavior.

More recently, threat monitoring has shifted towards cloud-native architectures and leveraging advanced analytics, including machine learning and behavioral analysis.

Cloud-based security platforms offer scalability and the ability to process vast amounts of data from distributed environments. Behavioral monitoring focuses on identifying deviations from normal user or system behavior, which is effective against zero-day threats and techniques that don't rely on known signatures. Furthermore, the integration of external **Threat Intelligence (TI)** feeds has become standard practice, enriching internal data with information about known bad actors, infrastructure, and attack techniques.

The Persistent Threat of Email

Despite the evolution of security technologies, email remains one of the most effective and frequently exploited vectors for initial access and malware delivery. Phishing attacks, designed to trick recipients into revealing sensitive information or executing malicious files, continue to be a primary concern. These include broad-spectrum phishing campaigns, highly targeted **Spear Phishing** attacks aimed at specific individuals, and sophisticated **Business Email Compromise (BEC)** scams that impersonate executives or trusted parties to initiate fraudulent wire transfers.

Email is also a prevalent channel for distributing malware. Malicious attachments (e.g., weaponized documents, executables) or links to malicious websites (malvertising, exploit kits, credential harvesting pages) are commonly delivered via email. Attackers constantly adapt their techniques to bypass traditional email security gateways, using social engineering, polymorphic malware, and legitimate cloud services to host malicious payloads.

Given this persistent threat, organizations require security postures that include specialized email monitoring capabilities that go beyond simple spam filtering or signature-based detection. While email gateways provide a necessary first line of defense, a dedicated system focused on email threats provides critical visibility into what *does* make it through initial filters or how sophisticated social engineering attempts are impacting users. This is where tools like the Threat Map application find their relevance – providing a focused view and management interface specifically for potential email-borne threats that require analyst review.

Core Use Cases in Modern Security Operations

Effective threat monitoring systems support a variety of critical use cases within organizational cybersecurity operations:

- **Intrusion Detection and Alerting:** By analyzing email data alongside other event logs, analysts can detect patterns indicative of an attempted or successful intrusion. For example, a phishing email attempting to harvest credentials might be correlated with suspicious login attempts originating from unusual locations. Monitoring systems generate timely alerts for security teams when specific threat indicators or suspicious activities related to emails (like a user clicking a known malicious link) are identified. The Threat Map application, by centralizing suspicious email data, provides the raw material for such detection and triage workflows, allowing analysts to quickly identify and prioritize emails tagged as high risk.
- **Threat Intelligence and Hunting:** Threat monitoring data provides valuable internal intelligence. By analyzing patterns of email threats targeting the organization (e.g., specific sender domains, subject lines, attachment types), security teams can identify active campaigns and understand attacker methodologies. This internal data can then be combined with external threat intelligence feeds to proactively hunt for threats that may have bypassed initial defenses or are currently dormant within the environment. Threat Map's ability to store and filter email threat data enables analysts to search for specific indicators (like a suspicious IP address found in email headers) and see if other emails share those characteristics, aiding in threat hunting activities.
- **Incident Response & Forensics:** When a security incident occurs (e.g., a data breach initiated via a phishing link), historical threat monitoring data is invaluable for incident response and forensic investigations. The logs and records collected by these systems provide a timeline of events, helping responders understand how the incident unfolded, identify the initial point of compromise (which is often an email), determine the scope of affected systems or users, and trace attacker activity. Threat Map's detailed records of suspicious emails, including summaries and associated IPs, serve as a key data source during investigations involving email as an attack vector. The ability to add remarks and mark activity status in Threat Map allows analysts to document their findings and actions, creating an audit trail crucial for post-incident analysis.
- **Compliance and Auditing:** Many industry regulations and compliance frameworks (e.g., HIPAA, PCI-DSS, GDPR) mandate specific requirements for security monitoring, logging, and incident handling. Threat monitoring systems help organizations meet these requirements by providing documented evidence

of monitoring activities, detected threats, and the steps taken to address them. The reports generated by Threat Map can contribute directly to compliance documentation, demonstrating due diligence in monitoring email threats and providing records of how incidents were investigated and managed. The user management features (like Admin Panel controls) also align with audit requirements for managing access to sensitive monitoring data.

- **Continuous Improvement of Defense:** Threat monitoring data offers crucial insights for improving the organization's security posture over time. By analyzing trends in detected threats (which types of emails are most successful at reaching users, what subject lines are common, which departments are targeted), security teams can identify weaknesses in existing controls (e.g., email gateway rules, web filters) and tailor their defenses more effectively. This data also informs employee security awareness training programs, allowing organizations to educate users about the specific types of email threats they are most likely to encounter. Threat Map's reporting features (like generating reports by sender, threat type, or date range) provide the aggregated data necessary to identify these trends and inform strategic decisions.

In summary, threat monitoring systems are not merely passive data collectors; they are active components of a comprehensive security strategy, facilitating detection, analysis, response, and continuous improvement. The Threat Map application focusses on a critical aspect of this domain—email threats—providing tailored capabilities for visualisation, tracking, and management that are essential for contemporary security operations teams managing the persistent threat of phishing and malware delivered via email.

System Overview: Threat Map Web Application

The **Threat Map** web application is a purpose-built, modular platform designed to empower organisations in the ongoing struggle against email-based cyber threats. Its primary objective is to provide a centralised, intuitive, and secure interface that enables both **security analysts** and **administrators** to efficiently monitor, investigate, and manage suspicious email activity from initial detection through remediation.

Key Objectives and Functionality

- **Visibility into Threat Landscape:** Aggregate real-time and historical data on suspicious or malicious emails targeting the organization, providing comprehensive situational awareness.
- **Analyst Workflow Support:** Facilitate triage, investigation, and annotation of threats, including adding remarks, updating statuses, and generating detailed incident reports.
- **Administrative Control:** Enable secure management of user accounts and allowed sender lists, maintaining tight access controls and adhering to principle of least privilege.
- **Actionable Reporting:** Generate ad-hoc or periodic summary reports tailored by sender, threat type, or activity, supporting compliance and strategic security decision-making.

Target User Personas

- **Security Analyst:** The primary day-to-day user, responsible for reviewing new threat entries, investigating suspicious emails, annotating cases, and escalating or closing threats. Analysts benefit from robust search, filter, and reporting tools as well as fine-grained access to incident history and details.
- **Administrator (Admin):** Typically security or IT professionals with elevated privileges. Admins manage user accounts, assign roles, curate the list of trusted senders, and oversee system integrity. Their controls are strictly separated to prevent unauthorized access to sensitive administration tasks.

Interface Walk-Through

- **Dashboard:** The default landing view provides a sortable, filterable table of all detected threats. Analysts can search by sender, filter by severity, or click to expand any entry for detailed metadata, event history, and remarks. Interactive

elements allow updating status, editing remarks, or viewing direct links to supporting evidence (such as attachments or IP addresses).

- **Reports:** The Reports module is dedicated to summarizing threat trends and supporting compliance. Users can generate new reports by criteria—such as date range, threat type, sender, or activity status—and download them as PDFs for sharing or archiving. An overview table lists existing reports, allowing rapid access and management.
- **Admin Panel:** Accessible only to authenticated admins, this section provides granular control over system users and trusted senders. Admins can add, edit, or remove user accounts, modify roles (analyst/admin), and curate allowed sender lists to control which emails are considered potentially safe. All sensitive actions are tightly gated with secure authentication checks.

Integration with Security Operations

Threat Map is positioned as an integral component of an organization's email defense ecosystem, aligning with operational needs for both rapid response and long-term risk management. By centralizing threat data and streamlining workflows, it helps security teams transition from reactive threat handling to proactive, analytics-driven decision making. The platform’s modular design ensures adaptability—allowing for future integrations with external threat feeds, incident response systems, and machine learning-based analysis as detailed in later sections.

Subsequent sections of this report will delve into the technical architecture, JavaScript implementation, UI/UX principles, and quality assurance processes that underpin Threat Map, providing a blueprint for extending, integrating, and maintaining a robust email threat monitoring capability tailored to organizational requirements.

JavaScript Implementation Breakdown

The client-side of the Threat Map web application is structured as a Single-Page Application (SPA) delivered via multiple HTML files, each primarily driven by a dedicated JavaScript file. This design modularizes the frontend logic, assigning specific responsibilities for the Dashboard, Reports, and Admin Panel modules to their respective scripts: index.js, reports.js, and admin.js. These scripts are the engine of the user interface, handling everything from secure session management and data presentation to complex user interactions and real-time updates. They orchestrate the fetching of data from backend APIs, dynamically build and update the user interface elements, manage interactive components like tables and modals, and implement features such as searching, filtering, sorting, and report generation workflows. The core philosophy behind their implementation is an event-driven design, where user actions trigger specific functions that interact with the backend and subsequently update the view.

Overall Structure and Shared Patterns

While each JavaScript file is responsible for a distinct application module, they share several foundational patterns and utility functions to maintain consistency and reduce redundancy. Key shared aspects include:

- Authentication Handling: All protected sections (Dashboard, Reports, Admin) begin by checking for a valid JSON Web Token (JWT) stored in the browser's local storage. A common mechanism (likely involving a helper function like getAuth() or direct token retrieval and validation) ensures that API calls include the JWT in the Authorization header. If a valid token is not present or has expired, the user is typically redirected to the login page.
- Role-Based Access Control (RBAC): The scripts implement frontend checks based on user roles embedded within the JWT (specifically, Cognito group claims). This determines, for example, whether the "Admin Panel" navigation link is displayed or if access to the admin.js-driven section is permitted. This frontend check complements backend API authorization managed by API Gateway and Lambda.
- Backend API Interaction: Interaction with AWS Lambda functions via API Gateway is central to all modules. Functions like fetch() (or a wrapper like

getAuth() which adds authentication headers) are used extensively for asynchronous data retrieval (GET) and for sending data or triggering backend processes (POST, PUT, DELETE).

- Dynamic UI Updates: JavaScript is used to manipulate the DOM based on fetched data and user actions. Functions like bodyBuild() are common across scripts for dynamically populating tables from data arrays. Placeholder states (e.g., "LOADING" skeletons) are shown while data is fetched and hidden upon successful rendering.
- Event Handling: The application relies heavily on event listeners attached to HTML elements (buttons, input fields, table rows, dropdowns) to respond to user input. This includes click, keyup, change, and keydown events, driving the application's interactivity.

admin.js - Admin Panel Logic

The admin.js script governs the Admin Panel, a privileged area for managing users and allowed sender lists. Its logic prioritizes security and administrative workflows:

- Initialization & Access Control: Upon loading, admin.js immediately checks for authentication and verifies the user's administrator status using the isAdmin() helper function which decodes the JWT. Unauthorized users are swiftly redirected to the login page, safeguarding sensitive administrative functions.
- Data Fetching: If access is granted, it triggers asynchronous calls to backend APIs to retrieve the list of application users (via a Cognito API) and the list of allowed email senders (from a DynamoDB table).
- UI Rendering: The fetched data populates two distinct HTML tables, dynamically constructed using the bodyBuild() function. Each row includes interactive elements (dropdown menus for users, trash icons for senders) to initiate edit or delete actions. A "LOADING" state is displayed until data is ready.
- Admin Actions (`adminAction`): This is the core function for handling administrative operations. It determines the specific action (Create User, Update User, Delete User, Add Sender, Delete Sender) based on the clicked button's identifier. It performs client-side validation to ensure required fields are filled before constructing the appropriate request payload (JSON body and URL parameters) and sending it to the relevant backend API endpoint (Cognito Users Modification API or Update Sender Data API) with the JWT for authentication.
- Dialog and Form Management: The script utilizes the HTML <dialog> element as a modal container for administrative forms (create user, edit user, delete confirmation, etc.). The dialogView(key) function controls which specific form

within the dialog is displayed based on the triggered action, managing their visibility via CSS. The showDialog() method displays the modal, and closeDialog() hides it and clears input fields to prevent data persistence.

- State Management: Data is fetched and stored in local variables. Actions triggered by the user result in API calls, and upon successful completion, the UI state is updated by re-fetching data and rebuilding the tables. This simple approach ensures the UI reflects the current state of the backend data.

index.js - Dashboard (Threats List) Logic

index.js powers the main Dashboard, where users view, search, filter, sort, and inspect details of detected threats. It is the most complex script in terms of UI interactivity and data manipulation:

- Initialization & Token Exchange: The script handles the post-login flow, checking for an authorization code in the URL to perform an OAuth token exchange with Cognito's token endpoint using exchangeCodeForToken(). The received ID and access tokens are stored in local storage for subsequent API calls. It then performs the standard token check and login redirect if necessary, and sets the Admin link visibility.
- Initial Data Load: fetchCompleteData() retrieves all threat records from a DynamoDB table via a backend API. The raw data, often in DynamoDB JSON format, is transformed into a cleaner JavaScript array structure. This data is stored in actualData (the full dataset) and filteredData (used for applying search/filter/sort), with filteredData initially a clone of actualData. The table is then built using bodyBuild(), and the loading indicator is removed.
- Table Rendering & Detail View Transition: bodyBuild() dynamically creates table rows for each threat entry, displaying key metadata. A click event listener is attached to each row. Clicking a row triggers a transition: it captures the threat data object, updates the URL query string with the threat's unique ID, and uses showLayout() to smoothly transition the UI from the main table view to the detailed threat view.
- Threat Detail View: The addDataView() function populates the detailed view with information from the selected threat, handling cases of missing data and formatting multi-line remarks stored with a custom delimiter (&&&). It also sets the text for the "Mark as active/inactive" toggle button. A "Return" button and an Escape key listener allow the user to transition back to the main table view using showLayout() in reverse.

- Searching, Filtering, and Sorting: These features provide analysts powerful ways to narrow down the threats list:
- Search: A keyup listener on the search input calls searchTable(), which filters the actualData (or filteredData if filters are active) based on a case-insensitive match in the 'from' field, then rebuilds the table with the result. HTML character escaping is included for basic sanitation.
- Filtering: Click listeners on filter buttons (by threat type) update a set of active filters. handleFilter() applies the active filters to the actualData array, updating filteredData, and rebuilds the table. Selecting "All" resets filters and refetches data for freshness.
- Sorting: A click listener on the "Date" column header toggles a sort order state (`data-order`). It then sorts the filteredData array client-side using JavaScript's built-in sort() method, converting date strings to Date objects for correct chronological sorting. A visual indicator (arrow icon) shows the current sort direction.
- Report Viewing & Remarks Editing: The script handles viewing pre-generated PDF reports (fetching a presigned S3 URL via API) and adding/editing analyst remarks. The remarks editor manages the conversion between display text (with newlines) and the stored format (using &&& delimiter). It includes basic input validation to prevent forbidden characters and uses a PUT request to an updateData API endpoint. The Save button is disabled during the API call to prevent duplicate submissions.
- State Management: Data is stored in local arrays (actualData, filteredData). UI state (which view is active, current filters/sort) is managed via CSS classes and DOM attributes. Actions that modify data trigger API calls, and success typically leads to re-fetching data to resynchronize the client state with the backend.

reports.js - Reports Page Logic

reports.js is dedicated to the Report generation and management interface:

- Initialization & Data Fetching: Similar to other scripts, it checks for authentication and admin status. It fetches two main datasets: the full threat metadata (email_metadata, from TABLE1 - the same data source as the Dashboard) and the list of previously generated reports (reportData, from TABLE2).
- Displaying Reports: The updateView() function checks if reportData is empty, showing either a "No reports found" message or the reports table. The bodyBuild() function populates the table with report entries, extracting names from S3 keys and displaying dates. Each row includes a downloadable icon

which, when clicked, calls downloadFile(). This function uses the get-pdf-url API to retrieve a presigned S3 URL and triggers the browser to download the PDF.

- Reports Table Filtering: A filter dropdown allows users to filter the displayed reports based on the report type (derived from the S3 key prefix). The script listens for change events on the dropdown, filters the reportData array, and rebuilds the table. Selecting "All" resets the filter and refetches the report list to ensure it's up-to-date.
- Report Generation Workflow: A modal dialog (<dialog id="gen-report">) allows users to define criteria for a new report (Report by type, Date Range, specific filters like Sender, Threat Severity, or Activity). The dialogView() function dynamically shows/hides relevant input fields based on the selected "Report by" type. The confirmGen handler validates all selected inputs, filters the in-memory email_metadata based on the chosen criteria, and if data is found, calls the backend generateReport API via a PUT request, including the filtered threat data as the request body and the JWT for authorization.
- State Management: Data is held in email_metadata and reportData arrays. The UI state (showing "No reports" or the table) is managed by updateView(). The report generation dialog state is managed by showDialog(), closeDialog(), and dialogView(). Upon successful report generation, the local reportData is refreshed by re-fetching from the backend, and the view is updated.

Design Decisions and Maintainability

The decision to use separate JavaScript files for each major application module (Admin, Dashboard, Reports) promotes a clear separation of concerns. Each script encapsulates the logic specific to its view, reducing complexity within individual files. While this structure leads to some duplication of utility functions (like token checks or table building patterns), it simplifies development within each module and makes it easier to debug or modify a specific part of the application without affecting others.

The event-driven model allows for a responsive user interface, as user actions directly trigger necessary updates. Dynamic DOM manipulation, while requiring careful handling to avoid performance issues with very large datasets, provides a seamless SPA-like experience without full page reloads. The use of HTML5 <dialog> elements for modals leverages native browser functionality for accessibility and standard modal behavior.

Data is primarily managed client-side in JavaScript arrays (actualData, filteredData, reportData, email_metadata). While simple for smaller datasets, this approach requires refetching data from the backend after modifications to ensure consistency, rather than relying on complex client-side state synchronization mechanisms. For a larger, production-scale application, a state management library or a more sophisticated data caching strategy might be considered.

Error handling often relies on basic alert() messages for user feedback and console.error() for developer debugging. While functional for an internship project, more robust error reporting and user notification patterns would be beneficial in a production environment.

Overall, the JavaScript implementation provides a functional and interactive frontend built on solid web development principles, with a clear structure that facilitates understanding and future enhancements within each distinct module.

User Interface/UX Design and Accessibility

The design and implementation of the User Interface (UI) and User Experience (UX) for the Threat Map application were guided by principles of clarity, efficiency, and accessibility, catering specifically to the needs of security analysts and administrators. Recognizing that these users require rapid access to critical information and the ability to perform tasks effectively under pressure, the UI was built to be intuitive, responsive, and easy to navigate. This section expands upon the design choices, element implementations, and accessibility considerations that define the Threat Map user interface.

Design Principles and Modular Layout

Threat Map adopts a modular design, logically separating functionality into distinct views: Dashboard, Reports, and Admin Panel. This separation, reflected not only in the JavaScript files (index.js, reports.js, admin.js) but also in the HTML structure (separate containers or pages), ensures that users are presented with information and controls relevant only to their current task. This reduces cognitive load and prevents interface clutter.

- **Modular View Separation:** Each module addresses a specific workflow—daily threat triage (Dashboard), historical analysis and summary (Reports), and system configuration (Admin). Transitioning between these views is handled via a consistent navigation bar at the top, which includes a logo, links to the main sections, and a user logout option.
- **Consistent Navigation:** The header navigation remains constant across accessible pages, providing a persistent anchor for users to move between modules. The conditional display of the "Admin Panel" link based on user role (verified via JWT claims) ensures that only authorized users see and can attempt to access administrative functions, adhering to the principle of least privilege from a UI perspective.
- **Single-Page Application Feel:** While implemented with separate HTML files for structural clarity during development, the application achieves a Single-Page Application (SPA) feel through dynamic content swapping within the main content area. For example, on the Dashboard, the main table view and the individual threat detail view are separate HTML containers (elements) that are shown

or hidden using CSS classes (like .active) and transitions. This technique, managed by JavaScript functions like showLayout(), provides smooth visual transitions (fading between views) without requiring full page reloads, resulting in a faster and more fluid user experience, particularly when drilling down into threat details or returning to a list.

- **Responsive Layout:** The layout is designed with responsiveness in mind, utilizing flexible CSS properties like Flexbox (implied by utility classes like display: flex and alignment classes) and relative units (percentage widths for table columns). This allows the interface to adapt to different screen sizes. While a complex data table view like the Dashboard is primarily optimized for larger screens (where horizontal scrolling might be necessary on smaller displays, as indicated by overflow-auto on table containers), the use of responsive units ensures core content remains accessible and usable across devices, from large monitors to tablets and potentially mobile phones (though the mobile experience for detailed data tables was a known limitation given the project scope).

Key UI Elements and Interactions

The Threat Map interface relies on standard, familiar UI patterns to guide users and present information effectively.

- **Data Tables:** Core data (threats, users, senders, reports) is presented in tabular format, which is ideal for organizing structured information. Tables include clear column headers, index numbers for easy reference, and visual indicators (like the up/down arrow next to the 'Date' column on the Dashboard) to show the current sort order. Clickable rows (on the Dashboard) provide a discoverable action to view details, reinforcing the visual structure with interactive functionality.
- **Forms and Dialogs:** User input for creating/editing data or generating reports is handled via forms displayed within HTML <dialog> elements. These native modal dialogs provide a standard and accessible way to present forms, centering them on the screen and blocking interaction with the background content. Forms within the dialogs follow a simple vertical stacking of labels and input fields for readability. Action buttons (e.g., "Cancel", "Create", "Update", "Confirm") are clearly labeled and positioned, often at the bottom of the form, with distinct styling (color, emphasis) to differentiate primary actions. The dynamic nature of forms, particularly in the Reports dialog where fields appear/hide based on user selections, streamlines the input process and reduces potential confusion.
- **Action Affordances and Feedback:** Actions are clearly presented through buttons, icons, or clickable elements. For example, download icons for reports,

trash icons for deletion (requiring confirmation), and distinct buttons for saving changes or confirming generation. Visual feedback is provided for user actions:

- **Skeleton Loading:** Placeholder "LOADING" states or skeleton effects are shown while data is being fetched, managing user expectations and preventing sudden shifts in layout once data arrives. This is removed (e.g., by removing a CSS class like skeleton) once the bodyBuild() function completes rendering.
- **Transitions:** Smooth CSS transitions are used for view changes (list to detail) and potentially other element states, making the interface feel more polished and responsive. The use of the transitionend event ensures actions dependent on the transition completion (like hiding the old view) are timed correctly.
- **Button States:** Interactive buttons, such as the "Save" button for remarks or the "Confirm Generate" button for reports, are disabled programmatically when an asynchronous operation (like an API call) is in progress. This prevents duplicate submissions and clearly indicates to the user that the system is processing their request.
- **Notifications and Alerts:** Simple browser alert() dialogs are used to provide feedback on action outcomes (e.g., "Report generated!!", "Operation successful!") or to flag validation errors ("Please fill all required fields.", "No records found."). While straightforward and reliable, future enhancements could explore more integrated, non-blocking notification systems.
- **Iconography:** Icons from Font Awesome are integrated throughout the application (e.g., gear/mask for logo, download arrow, trash can). These visual cues quickly communicate the purpose of interactive elements and enhance the interface's scannability, making it easier for users to identify actions at a glance.

Accessibility Features

Accessibility was considered during the design and implementation process to ensure the Threat Map application is usable by individuals with diverse needs, including those using assistive technologies like screen readers or relying on keyboard navigation.

- **Semantic HTML:** The use of semantic HTML5 elements (<header>, <main>, <nav>, <h2>, <ul>, <table>, <dialog>, <form>, <button>, <label>) provides inherent structure and meaning to the content, which is crucial for screen readers to interpret and convey the page structure to users. Headings (H2, H3) correctly outline the document hierarchy.
- **Labeling and Alt Text:** All form input fields are correctly associated with descriptive <label> elements using the for attribute referencing the input's id. This ensures that screen readers announce the purpose of each input when a user focuses on it. Similarly, meaningful images or icons (like the "No reports found" illustration) include appropriate alt text (e.g., alt="no reports") to provide a textual alternative for users who cannot see the image. Interactive icons, even when part of buttons or links, should ideally have associated accessible names (via text content, aria-label, or title attributes, although explicit ARIA attributes beyond standard element behavior were not extensively detailed in the provided code).
- **Keyboard Navigation:** Standard interactive HTML elements like buttons, links, input fields, and select dropdowns are inherently focusable and operable using keyboard commands (Tab, Shift+Tab, Enter, Space). This baseline keyboard support is essential for users who cannot use a mouse. Additionally, an explicit keyboard shortcut (Escape key) was implemented to exit the threat detail view, providing a convenient alternative to clicking the 'Return' button and demonstrating attention to keyboard-only user workflows. The use of the native HTML <dialog> element also assists with accessibility by automatically managing focus within the modal when it's open and returning focus to the trigger element when closed.
- **Readability and Contrast:** While specific color palettes are not detailed, the description of a "minimalist black/white/gray theme" and the use of dark text on light backgrounds (implied by standard text color and dividing lines) suggests an effort towards sufficient contrast for readability. Clear, straightforward language is used in UI text (labels, buttons, messages) to minimize confusion.

User Feedback and Lessons Learned

Although formal user testing with external individuals was beyond the scope of the internship, iterative development and extensive internal testing (as detailed in the Testing and Quality Assurance section) implicitly incorporated a form of user feedback by identifying and addressing usability issues.

- **Learning from Testing:** Issues discovered during testing, such as the initial case-sensitive search or the failure to preserve line breaks in remarks, directly informed UI/UX improvements. Fixing these problems highlighted the importance of anticipating common user behaviors (like typing search terms without regard for case or expecting multi-line text areas to handle newlines) and implementing robust handling for data input and display.
- **Simplicity Aids Usability:** The decision to stick to fundamental HTML elements and a relatively simple layout proved beneficial. For users who need to quickly scan data and perform specific actions, a clear, tabular interface with predictable controls (buttons, dropdowns, modals) is highly usable. Overly complex custom UI widgets or novel interaction patterns might have introduced a steeper learning curve.
- **Visual Feedback is Crucial:** Implementing clear visual feedback mechanisms, such as the skeleton loading, button disabling during processes, and confirmation alerts, was learned to be vital. Users need to know when the system is working, whether their action succeeded, or if there was an error, especially in an asynchronous web application interacting with backend APIs.
- **Client-Side Validation Enhances UX:** Adding client-side checks for required form fields or logical constraints (like ensuring the report 'From' date is not after the 'To' date) provides immediate feedback to the user before sending data to the backend. This saves time and prevents unnecessary API calls and server errors, improving the user experience.
- **Maintaining State Consistency:** A key lesson was the importance of ensuring the UI accurately reflects the backend state after any data modification. The pattern of re-fetching and rebuilding tables after actions like creating/deleting users or generating reports, while sometimes appearing basic, reliably ensures the user interface is synchronized, preventing display errors or stale data issues.

Threat Map’s UI/UX design prioritised functionality, clarity, and accessibility. Modular structure, consistent navigation, and dynamic content presentation streamlined workflows. Standard UI elements, action feedback, and foundational accessibility features ensured usability for the target audience. Iterative testing reinforced the value of user-centric design in technical project prototypes.

System Architecture and Data Flow

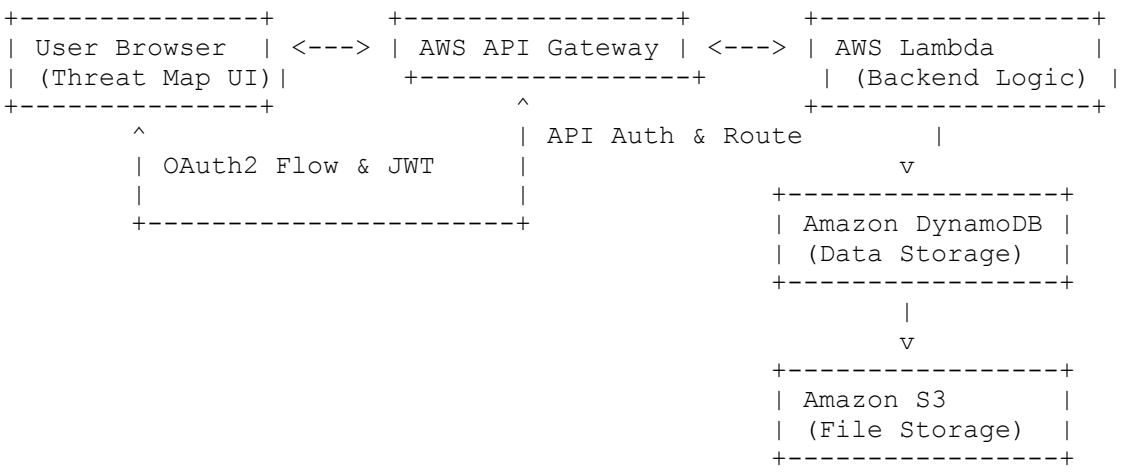
The Threat Map application is underpinned by a robust, serverless architecture built primarily on Amazon Web Services (AWS). This design choice provides inherent advantages in terms of scalability, cost-efficiency (pay-as-you-go model), reduced operational overhead (no server management), and security (leveraging managed services). The architecture follows a clear separation of concerns, with the client-side UI interacting with a set of backend services exposed through an API Gateway, handling authentication, data storage, compute logic, and file management.

AWS Service Components

The core backend comprises the following AWS services:

- **AWS Cognito:** Manages user authentication and authorization. It acts as the identity provider, handling user registration, login, and issuing JSON Web Tokens (JWTs) that contain user identity and group membership (e.g., 'Admin' role).
- **AWS API Gateway:** Serves as the secure front door for all backend API requests originating from the Threat Map UI. It routes incoming requests to the appropriate backend service (primarily AWS Lambda functions) and enforces authentication and authorization using a Cognito Authorizer.
- **AWS Lambda:** Provides the serverless compute environment for executing backend logic. Each distinct API operation (fetching data, managing users, generating reports, etc.) is typically implemented as a separate Lambda function. Lambdas interact with other AWS services like DynamoDB and S3.
- **Amazon DynamoDB:** A fully managed NoSQL database service used for storing structured data such as threat records, report metadata, and allowed sender lists. Its key-value and document data model is well-suited for storing the varied data structures used by the application.
- **Amazon S3 (Simple Storage Service):** Provides scalable and durable object storage. In Threat Map, it is primarily used for storing larger, static files like generated PDF reports and potentially original email files or attachments associated with threats.

Below is a conceptual diagram illustrating the high-level interaction between these components and the user interface:



This diagram highlights the flow where user requests initiated in the browser go through API Gateway, triggering Lambda functions that interact with DynamoDB for data persistence and S3 for file storage. AWS Cognito integrates with both the browser (for the initial login flow) and API Gateway (for request authorization).

Component Configuration and Security

Implementing this architecture involves specific configurations for each AWS service to ensure functionality, security, and scalability:

- **AWS Cognito:**
  - **User Pool:** Configured to manage user accounts. Includes attributes like username, email, and custom attributes if needed. Email verification is typically enabled for security. Password policies are set according to security best practices.
  - **App Client:** Created for the web application, allowing it to interact with the User Pool. It is configured with allowed OAuth flows (e.g., Authorization Code Grant with PKCE for web apps), callback URLs (redirecting back to the Threat Map UI after login/logout), and allowed OAuth scopes (e.g., openid, email, profile).
  - **User Groups:** Groups like 'Admin' and 'User' are defined within the User Pool. Users are assigned to these groups, and this membership is included in the JWT's claims (specifically, the cognito:groups claim). This enables Role-Based Access Control.

- **Domain:** A hosted UI domain is configured for the User Pool, providing standard login/signup pages managed by AWS, reducing the need for custom-built authentication forms and improving security.
- **Cognito Authorizer (in API Gateway):** Configured to validate the JWT presented in the Authorization: Bearer header of incoming API requests against the User Pool. It checks the token's signature, expiry, and issuer. This ensures that only authenticated and valid requests reach the backend Lambda functions. API Gateway resources (endpoints) can be configured to require this authorizer, enforcing authentication at the edge. Furthermore, specific API endpoints can be configured to require certain claims or group memberships (e.g., only allow requests to /admin/* endpoints if the JWT contains the 'Admin' group claim), enforcing authorization.
- **Security:** Cognito handles password hashing, multi-factor authentication (MFA) support (if enabled), and rate limiting against brute-force attacks on the hosted UI. Communication is always via HTTPS. The separation of authentication concerns into Cognito reduces the application's attack surface.

- **AWS API Gateway:**
  - **REST API:** Configured with resources and methods corresponding to the application's backend functions (e.g., /fetchFromDynamo, /admin/users, /generateReport, /get-pdf-url).
  - **Integration Type:** Typically configured with Lambda proxy integration, which passes the entire request context (headers, body, path parameters, query strings) to the Lambda function, giving the Lambda full control over processing the request and formatting the response.
  - **CORS Configuration:** Necessary to allow the browser-based Threat Map UI (served from a different origin/domain than the API Gateway endpoint) to make requests to the API. Proper CORS headers (Access-Control-Allow-Origin, Methods, Headers) are configured at the API Gateway level.
  - **IAM Permissions:** API Gateway requires permissions to invoke the target Lambda functions. This is managed via IAM roles.
  - **Security:** Enforces HTTPS for all communication. The Cognito Authorizer provides robust authentication and authorization. Usage plans and API keys can be added for monitoring and rate limiting, though perhaps less critical for an internal internship project.
  - **Scalability:** API Gateway is a managed service that automatically scales to handle varying levels of request traffic.
- **AWS Lambda:**

- **Functions:** Each Lambda is written in a supported runtime (e.g., Node.js, Python, Java) and contains the specific logic for a backend operation. Configuration includes memory allocation (impacts performance and cost), timeout settings (to prevent runaway functions), and environment variables (for configuration like database table names or API endpoints).
- **IAM Role:** Each Lambda function is assigned an IAM execution role. This role defines the permissions the Lambda function has to interact with other AWS services (e.g., read/write access to specific DynamoDB tables, put/get object access to specific S3 buckets, permissions to call Cognito Admin APIs). This adheres to the principle of least privilege – a function handling threat data doesn't need permission to modify user accounts, for example.
- **Networking:** For accessing resources within a VPC (like RDS or potentially other internal services), Lambdas would need to be configured within the VPC. However, for accessing managed services like DynamoDB, S3, and Cognito endpoints (which are publicly accessible AWS endpoints), VPC configuration is not strictly necessary unless specific networking requirements exist. Communication with these services is via secure AWS SDK calls over the internet (secured by TLS).
- **Scalability:** Lambda automatically scales the number of function instances based on the volume of incoming requests. This provides high availability and the ability to handle sudden spikes in traffic without manual intervention. Cold starts (latency on the first invocation after a period of inactivity) can be a factor but are often mitigated by provisioned concurrency for critical functions or managed warm-up strategies.

- **Amazon DynamoDB:**
  - **Tables:** At least three tables are defined: Threats (TABLE1), Reports (TABLE2), and Senders (TABLE3).
    - **Threats Table:** Likely uses a unique identifier (e.g., emailUid) as the Partition Key. Attributes include sender, subject, date, campaign, type, suspect_ip, summary, remarks, isActive, s3Key (for related files), etc. For query efficiency, Secondary Indexes (Global or Local) might be considered on frequently queried attributes like 'date', 'type', or 'campaign' if client-side filtering isn't sufficient for large datasets.
    - **Reports Table:** Might use a Partition Key based on report type and/or a date/timestamp, or simply an S3 key identifier. Attributes would include the S3 key (filename), creation date, user who generated it, and potentially the criteria used for generation.

        - **Senders Table:** Likely uses the email address as the Partition Key for quick lookup and management.
    - **Consistency:** DynamoDB offers Eventual Consistency (default) and Strong Consistency. Reads are eventually consistent by default, meaning a read might not reflect the result of a recently completed write. Strong consistency ensures reads return the most up-to-date data but have higher latency and cost. In Threat Map, for flows like refreshing tables after an update, eventual consistency is generally acceptable, or small delays are introduced client-side, assuming the brief period of inconsistency is not critical for user experience. For administrative actions where immediate confirmation is needed (e.g., verifying a user was created), a strongly consistent read might be preferable if critical logic depends on it, although the current implementation likely relies on re-fetching and rebuilding the UI, implicitly handling eventual consistency.
    - **Scalability:** DynamoDB is highly scalable and can handle massive amounts of data and traffic. Provisioned throughput or On-Demand capacity modes allow adjusting capacity based on predictable or unpredictable workloads.
    - **Security:** Access control is managed exclusively through IAM policies attached to Lambda execution roles, ensuring only authorized backend code can perform operations on the tables. Data can be encrypted at rest.
- **Amazon S3:**
    - **Buckets and Folders:** Dedicated S3 buckets are created (e.g., threat-map-reports). Folders (e.g., reports/, potentially emails/ or attachments/) are used to organize files.
    - **Access Control:** IAM policies control which Lambda functions (via their roles) can put or get objects in the bucket. To allow the browser to download reports without direct access to the bucket, Lambdas generate time-limited **presigned URLs**. These URLs grant temporary permission to download a specific object via HTTP GET directly from S3, expiring after a configurable duration, which is much more secure than making the bucket or objects publicly readable.
    - **Scalability and Durability:** S3 provides eleven nines (99.999999999%) of data durability and is designed for virtually unlimited storage capacity, automatically scaling to meet demand.
    - **Security:** Data is encrypted at rest by default. Access is controlled via IAM roles and presigned URLs. Versioning can be enabled to prevent accidental deletion or overwriting of files.

Key Data Flows Detailed

Authentication Flow

Authentication begins when an unauthenticated user attempts to access a protected page.

1. The Threat Map UI checks for a valid JWT in local storage. If none is found or it's invalid/expired, the user is redirected to the AWS Cognito Hosted UI login page.
2. The user enters credentials on the Cognito page.
3. Cognito authenticates the user and redirects the browser back to the Threat Map UI's configured callback URL, including an authorization code as a URL parameter.
4. The Threat Map UI (e.g., index.js on load) detects the authorization code. It sends a POST request to the Cognito token endpoint (via the frontend's exchangeCodeForToken() function) including the code, client ID, redirect URI, and grant type.
5. Cognito validates the code and client details, then issues ID, Access, and Refresh Tokens.
6. The UI receives the tokens and stores the id_token (used for authentication) and access_token (used by the frontend to call AWS services directly if needed, though not primary here) in browser local storage. A small delay might be added to ensure storage is complete before proceeding.
7. For subsequent API calls (e.g., fetching data, performing admin actions), the UI includes the id_token in the Authorization: Bearer [token] header using a wrapper like getAuth().
8. AWS API Gateway receives the request and the JWT. Its Cognito Authorizer validates the token against the User Pool. If valid and authorized (e.g., checks group claims), API Gateway invokes the target Lambda function. If validation or authorization fails, API Gateway rejects the request with a 401 Unauthorized or 403 Forbidden response before it even reaches the Lambda.
9. The Lambda function executes, knowing the request was authenticated and authorized by API Gateway. It can optionally inspect the token claims passed by API Gateway to implement more granular in-application logic (e.g., showing/hiding UI elements based on role as done in the frontend).
10. If a backend API call returns a 401 response (e.g., due to token expiry detected by the backend itself, though typically handled by Authorizer), the frontend client-side error handling detects this and triggers a logout flow (clearing local storage and redirecting to login).

Data Ingestion Flow (Poller)

Threat data enters the system via an ingestion mechanism, currently triggered manually.

1. A user clicks the "Refresh Email Metadata" button on the UI, which triggers an API call (HTTP GET or POST) to a specific API Gateway endpoint (e.g., /pollerV2).
2. API Gateway validates the request using the Cognito Authorizer.
3. API Gateway invokes the Poller Lambda function.
4. The Poller Lambda executes its logic to fetch new threat data. In a real-world scenario, this Lambda would connect to an external source like an email security gateway API, a security log aggregation system, or scan a dedicated mailbox.
5. The Lambda processes the raw data, extracts relevant metadata for each suspicious email, and formats it.
6. For each new threat record, the Lambda uses the AWS SDK to write an item to the Threats DynamoDB table (TABLE1). It handles potential duplicates based on a unique identifier.
7. The Poller Lambda responds to API Gateway (e.g., with a success message or count of new records).
8. API Gateway returns the response to the UI.
9. The UI, upon receiving a successful response, then triggers a standard data retrieval flow to fetch the updated list of threats and refresh the dashboard table, making the newly ingested data visible to the user.

Report Generation Flow

Report generation involves filtering data, creating a file, and recording metadata.

1. The user selects report criteria (type, date range, filters) in the Reports page dialog.
2. The UI validates the criteria and filters the local email_metadata array based on the user's selections, creating a subset of data (required_Metadata). This client-side filtering can be done server-side in the Lambda for robustness or to handle large datasets that don't fit in browser memory. Assuming the current client-side approach is used for criteria application, the relevant data rows are identified.
3. The UI sends an HTTP PUT request to the /generateReport API Gateway endpoint. The request body contains the filtered threat data (required_Metadata array), and the request includes parameters like the report type (e.g., query parameter by=sender). The JWT is included in the Authorization header.

4. API Gateway validates the token via the Cognito Authorizer and invokes the Generate Report Lambda function.
5. The Lambda receives the request body containing the threat data and the report type parameter.
6. The Lambda uses a PDF generation library (e.g., a headless browser rendering HTML, or a dedicated PDF library) to create a PDF document based on the provided threat data and report type formatting.
7. The Lambda generates a unique key (filename) for the PDF (e.g., reports/threat_analysis_2023-10-27.pdf).
8. The Lambda uses the AWS SDK to upload the generated PDF file to the configured S3 bucket and folder.
9. Upon successful S3 upload, the Lambda writes a new item to the Reports DynamoDB table (TABLE2), including the S3 key (filename) and creation timestamp.
10. The Lambda responds to API Gateway with a success status (e.g., HTTP 200).
11. API Gateway returns the success response to the UI.
12. The UI receives the success confirmation, displays a message to the user, closes the dialog, and triggers a data retrieval flow for the Reports table (fetching data from TABLE2).
13. The Reports table UI is rebuilt with the latest data, including the newly generated report, which is now available for download.

User and Sender Management Flow (Admin)

Admin actions involve modifying data in Cognito or DynamoDB.

1. An administrator user interacts with a form in the Admin Panel (e.g., Create User, Edit User, Delete Sender). The UI validates the input client-side.
2. Upon confirming an action, the UI calls the adminAction() function which determines the specific operation and constructs the appropriate payload and target API endpoint (e.g., PUT to /admin/cognito-users-modification?key=create, DELETE to /admin/updateSenderData). The JWT is included in the Authorization header.
3. API Gateway receives the request. The Cognito Authorizer validates the token and checks if the user belongs to the 'Admin' group (as the /admin/* path is restricted). If authorized, it invokes the corresponding Lambda function (e.g., User Management Lambda or Sender Management Lambda).
4. The Lambda function executes. Based on the request path, parameters, or body, it performs the required operation:

    - For user actions, it uses the AWS SDK to interact with the Cognito User Pools API (e.g., AdminCreateUser, AdminUpdateUserAttributes, AdminDeleteUser).
    - For sender actions, it uses the AWS SDK to perform put or delete operations on the Senders DynamoDB table (TABLE3).
5. The Lambda function handles the operation and returns a response (success or error) to API Gateway.
6. API Gateway returns the response to the UI.
7. The UI receives the response, shows feedback to the user (alerting success or failure), closes the dialog (if applicable), and triggers a data retrieval flow for the relevant table (Users or Senders) to refresh the displayed list from the backend, reflecting the changes.

Error and Exception Handling

Error handling is implemented across both frontend and backend layers:

- **Frontend:** JavaScript uses try...catch blocks around asynchronous API calls to handle potential network errors or non-2xx HTTP responses. Validation errors (e.g., empty required fields, incorrect date ranges) are caught client-side before sending requests and notified via alert() messages. HTTP 401/403 responses from API calls trigger session expiry handling (logout).
- **Backend (Lambda):** Lambda functions include logic to catch errors during interactions with DynamoDB, S3, Cognito APIs, or internal processing errors. Errors should be logged (e.g., using console.error or a logging library) to AWS CloudWatch Logs for monitoring and debugging. Appropriate HTTP status codes (e.g., 400 for bad request, 500 for internal server error) and informative error messages are returned to the frontend.
- **API Gateway:** Configured to return standard HTTP error responses (e.g., 403 Forbidden from Authorizer, 502 Bad Gateway if Lambda fails). Custom error mapping can be configured for more specific responses.

Deployment and DevOps Considerations

Deploying and managing this serverless application involves several DevOps practices:

- **Infrastructure as Code (IaC):** Defining the AWS resources (API Gateway, Lambdas, DynamoDB tables, S3 buckets, Cognito User Pool, IAM roles) using tools like AWS CloudFormation, AWS SAM (Serverless Application Model), or HashiCorp Terraform. This allows for repeatable, version-controlled infrastructure deployments across different environments (dev, staging, prod).
- **CI/CD Pipelines:** Implementing Continuous Integration and Continuous Deployment pipelines using services like AWS CodePipeline, CodeBuild, and CodeDeploy (or third-party tools like GitHub Actions, GitLab CI). The pipeline would automate:
  - Building the frontend code (e.g., running linters, bundling JavaScript/CSS).
  - Deploying the frontend assets to S3 and invalidating CloudFront cache (if used).
  - Building and packaging Lambda function code and dependencies.
  - Deploying the backend Lambda functions and API Gateway configuration using the IaC template.
  - Running automated tests (unit, integration).
- **Versioning:** Managing versions of Lambda functions and API Gateway stages (e.g., /prod, /dev) to allow for safe deployments and rollbacks. S3 also supports object versioning for files like reports.
- **Monitoring and Logging:** Utilizing AWS CloudWatch for centralized logging of Lambda executions and API Gateway access logs. Setting up CloudWatch Alarms on key metrics (e.g., Lambda error rates, duration, throttles; API Gateway 5xx error count) to be notified of operational issues. Distributed tracing (e.g., using AWS X-Ray, though potentially complex for a simple project) can help diagnose performance bottlenecks across services.
- **Secrets Management:** Storing any sensitive configuration data (e.g., API keys for external threat intelligence feeds if integrated later) in AWS Secrets Manager or Parameter Store instead of hardcoding them in Lambda environment variables.

While the internship project may not have implemented a full CI/CD pipeline or extensive monitoring, these are crucial considerations for transitioning to a production environment, ensuring reliability, security, and maintainability of the serverless architecture.

Detailed Functional Walkthroughs

Authentication and Login Workflow

Secure access and streamlined user onboarding form the foundation of the Threat Map experience. User authentication is handled via AWS Cognito, providing OAuth2-compliant flows, JWT issuance, and robust group-based role enforcement.

- **Initial Access:** On first navigation to a protected Threat Map page (Dashboard, Reports, Admin), the JavaScript initializes by checking for an id_token in local storage.
  - If absent or expired, the user is **redirected** to the Cognito Hosted UI login page.
- **Login & Token Exchange:** The user authenticates via the Cognito UI. Upon success, Cognito redirects back to the configured callback URL, appending an authorization code. The frontend uses exchangeCodeForToken() to exchange this code for id_token and access_token, which are then **persisted in local storage**.
  - *Error Case:* If the token exchange fails due to network or credential issues, an alert is shown, and the page redirects to the login screen.
- **Role Determination:** The id_token is decoded client-side to extract Cognito group claims, determining if the user is an **Admin** or **Analyst**. Admin-only functions (such as the Admin Panel link) are conditionally rendered in the navigation bar.
- **Session Management:** If a token expires (e.g., due to inactivity), protected API calls will return 401 responses. JavaScript catches this and triggers an automatic logout—clearing tokens, alerting "Session expired. Please log in again.", and redirecting to login.

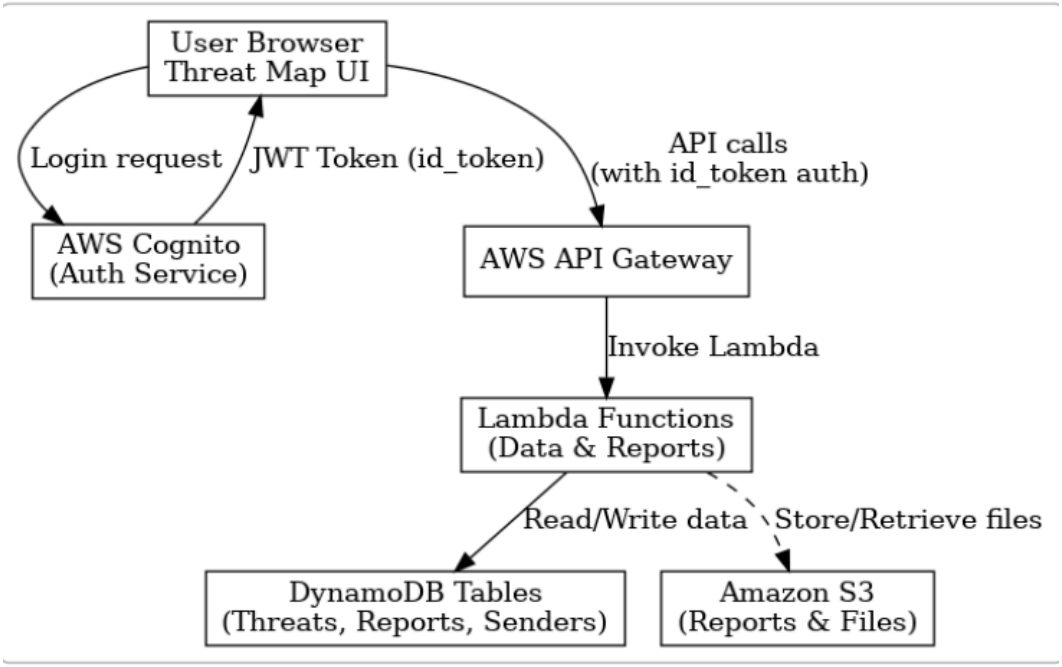


Figure 1: Simplified authentication and state transitions for the Threat Map UI

Viewing Threats: Dashboard Table Operations

1. **Loading Threat Data:**
   - Upon successful authentication, fetchCompleteData() retrieves all threat records from the backend, displaying a "LOADING" skeleton until JSON parsing and DOM updates complete.
   - Error handling provides "No Records Found" if no data is returned or parsing fails.
2. **Tabular Display:**
   - The main Dashboard view displays threats as rows with columns: #, From, Campaign, Type, Activity, Date.
   - Clicking on a row transitions to a corresponding detail view.
3. **Table Interactivity & Feedback:**
   - Rows are made keyboard-focusable (via tabindex=0) for accessibility.
   - Visual indicators (e.g., hover highlight) guide the user for clickable rows.

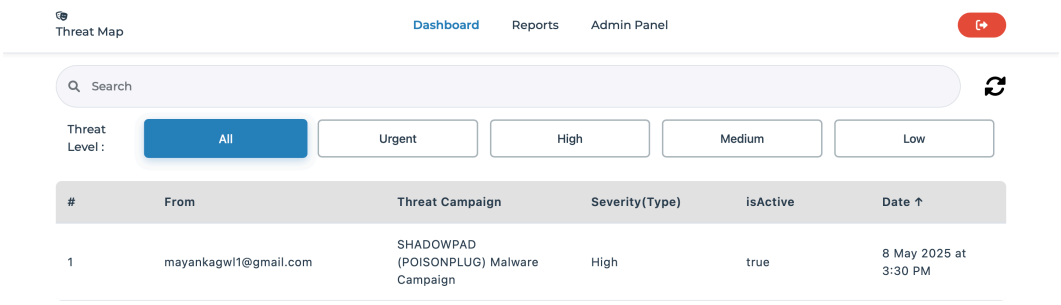


Figure 2: Annotated Dashboard table with filter, search, and sort controls.

Searching, Filtering, and Sorting Threats

- **Searching:**
  - Entering a string in the table search box (by sender email) triggers searchTable() on each keyup event.
  - Case-insensitive matching displays only rows with senders containing the input substring.
  - *Edge Case:* No matches found shows "No Records Found" feedback. Special character input is sanitized to prevent injection.
- **Filtering by Severity:**
  - Clicking the filter buttons (Urgent, High, Medium, Low) invokes handleFilter(), modifying the active filter set. Multiple severity types can be active at once.
  - Selecting 'All' resets filters and triggers a server-side refresh.
  - The filtered data array is used for subsequent search/sort actions.

- **Sorting by Date:**
  - Clicking the "Date" column header toggles between ascending and descending order, indicated by an arrow icon.
  - If rows are filtered or searched, sorting applies only to the displayed subset.

Threat Detail View and Review Workflow

1. **Transition to Detail View:** When a row is selected, the corresponding threat's data object is stored and addDataView() populates all fields in the detail panel. The UI transitions with a fade-out/fade-in managed by showLayout() and CSS.
2. **Displayed Details:**
   - Fields include: Sender, Subject, Campaign, Severity (color-coded), Date, Suspect IP(s), Activity status, Summary, and Analyst Remarks.
   - Each field handles missing data gracefully (e.g., displaying "Not Available" if a value is empty).
3. **Returning:** "Return" button or pressing Escape returns to the table, retaining active filters/search.
4. **Feedback Patterns:**
   - On invalid transitions (e.g., missing detail data due to a direct link to a stale/nonexistent threat), a warning is shown and user is redirected back to table.

Editing Analyst Remarks

1. **Initiate Edit:** In the detail view, clicking "Edit Remark" slides open a textarea pre-filled with the current remarks, converting the &&& delimiter to newlines.
2. **Validation & Save:**
   - On Save, the input is checked for forbidden characters (<, >) and empty content. The Save button is disabled during the API call.
   - If validation fails, an alert is shown; input is not sent to server.
   - On success, the backend updates the threat in DynamoDB, and the UI receives confirmation, updates the display, and refreshes the filtered data.

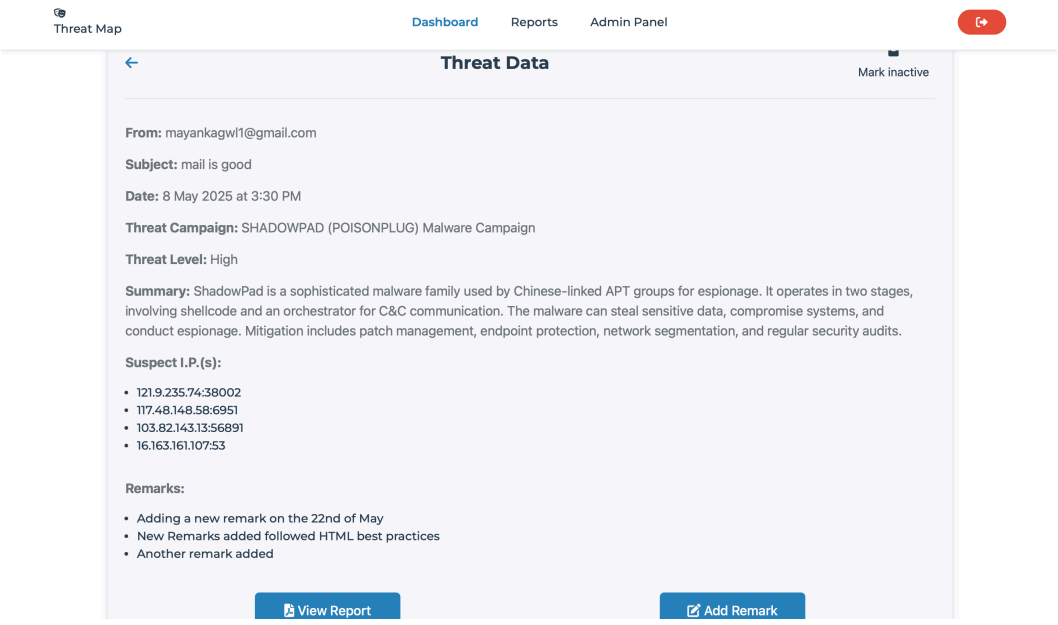


Figure 3: Expanded threat detail panel, showing full metadata and remarks.

3. **Canceling Edit:** Clicking Cancel or losing focus closes the editor without saving changes.
4. **Error Handling:** Network or server errors in the save step result in an error alert, re-enabling the Save button for retry.

```
// Error feedback when disallowed characters are present
if (remarksInput.includes('<') || remarksInput.includes('>')) {
    alert("Remarks contain forbidden characters < or >.");
    return;
}
```

Generating and Downloading Reports

1. **Generate New Report:**
   - User clicks "Generate Report" (from empty state or table view) to open a modal dialog. Dropdowns and date pickers guide parameter selection (type, sender, severity, activity, date range).
   - On Confirm, client-side checks require all necessary fields; date ranges are validated to ensure From ≤ To. Invalid conditions prompt a corrective alert.
   - Upon valid submission, the in-memory dataset is filtered; if matches exist, the data is sent to the backend via secure PUT request to the generateReport API.
2. **Download Workflow:**
   - Newly generated reports appear in the table instantly without refresh.
   - Clicking the download icon triggers API call for a one-time, time-limited S3 presigned URL; the file opens or downloads in-browser.
   - Failed downloads (expired URL, unavailable file) prompt an error feedback, suggesting retry or contacting admin.

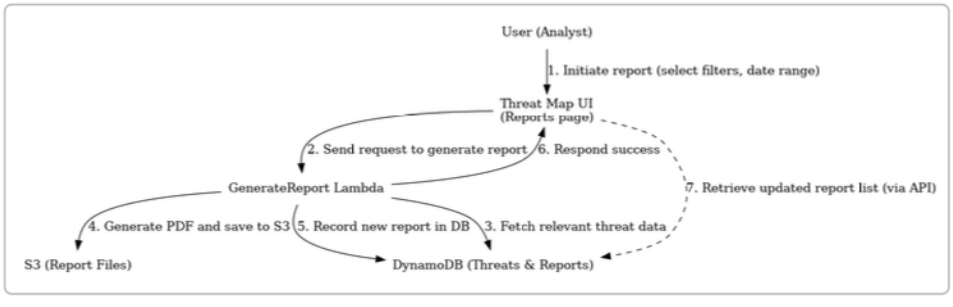


Figure 4: Report generation workflow

3. **Filtering and Searching Reports:**
    - Dropdown filters by report type instantly adjust the visible list. Resetting to 'All' always refetches from the backend for data freshness.

Admin Panel: User and Sender Management

1. **Access Control:** Only users with the "Admin" group claim can access or see the Admin Panel link. Unauthorized attempts result in redirect to login, with an alert. The isAdmin() helper performs client-side checks on page load.
2. **User Management:**
    - **Creating Users:** "Create User" opens a modal; all fields (username, email, temp password, role) are required. Form validation prevents empty submissions.
    - **Editing Users:** "Edit" opens the form pre-filled with current data. Admins can modify email and status (enabled/disabled). Changes are submitted via API, with immediate feedback on success or specific errors (e.g., email already exists).
    - **Deleting Users:** "Delete" invokes a confirmation dialog. Confirmed deletions update the table post-backend removal.
3. **Sender Management:**
    - **Adding Senders:** Admin inputs a new sender email, which is validated for format and required status. Success feedback instantly updates the "Allowed Senders" table.
    - **Removing Senders:** Trash icon prompts a confirmation dialog. Confirming removal updates the server and the UI; table reindexes as needed.
    - *Error Case:* Attempting to delete a non-existent sender or upon backend error triggers an alert specifying the issue.
4. **General Feedback Patterns:**
    - All successful operations display brief confirmation alerts (e.g., "Operation successful!").
    - Form fields are reset after each action to prevent residual values.
    - Dialogs can be canceled at any time, discarding unsaved changes and closing the modal.

Robust Error Handling and User Feedback

- **Field Validation:** All forms require mandatory fields before allowing submission. Validation errors prompt explicit alerts.
- **Session Expiry:** If tokens expire or are purged, the user is alerted and redirected, preventing unauthorized access.
- **Backend and Network Failures:** API failures are logged to the console and prompt user-friendly error alerts, distinguishing temporary network errors from persistent backend issues.
- **UI Consistency:** Tables and views are always refreshed after any mutation action to reflect the backend state, eliminating stale or inconsistent client-side data.
- **Keyboard Accessibility:** All critical workflow actions (table navigation, dialog interaction, form submission/cancellation) are fully accessible via keyboard, enhancing usability for all user profiles.

Summary State Diagram

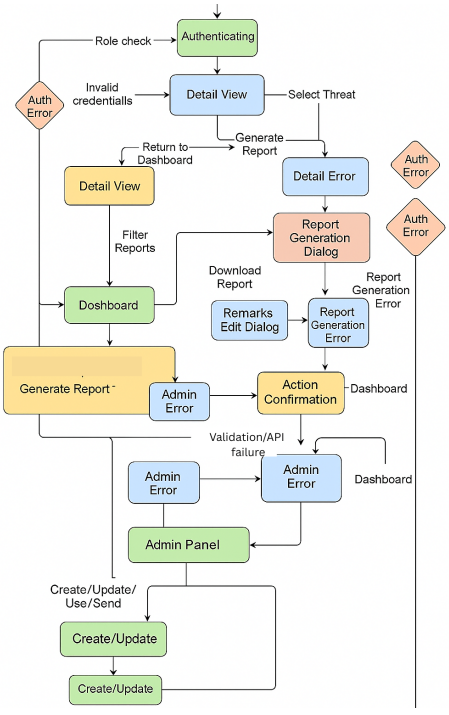


Figure 5: Comprehensive state diagram showing primary frontend workflow states, transitions, and error/feedback nodes across authentication, Dashboard, detail, report, and Admin Panel modules. image

Intentional Simplifications and Future Inspirations

Threat Map was designed with specific constraints and goals as an internship project. Its simplifications are intentional reflections of this scope:

- **Focused Vector:** Limiting the scope to email threats made the project manageable, allowing for a deeper dive into a specific, critical area rather than a broad but shallow implementation across all threat types.
- **Manual/Scheduled Ingestion:** The reliance on a manual or periodic trigger for data ingestion simplifies the backend architecture compared to building a robust, event-driven real-time data pipeline.
- **Client-side Processing:** Features like filtering and sorting on the Dashboard are largely handled client-side after fetching all data. While performant for moderate datasets, this is simpler than implementing complex server-side querying or dedicated search engines needed for massive data volumes.
- **Basic UI Feedback:** Using standard browser alerts for notifications is a simple, effective way to provide feedback without needing to build a custom, complex notification system.

These simplifications allowed the project to successfully demonstrate core concepts like secure cloud architecture, data visualization, role-based access, and basic workflow management within a defined timeline. However, the comparison with real-world tools provides a clear roadmap for expansion. The proposed future integrations—adding API hooks for external data, incorporating threat intelligence feeds, integrating sandboxing for dynamic analysis, and applying machine learning for automated classification and prioritization—are directly inspired by the advanced capabilities observed in tools like VirusTotal, FireEye, and Microsoft Defender. These enhancements would enable Threat Map to move beyond a basic monitoring dashboard towards a more intelligent, integrated, and operationally valuable threat analysis platform.

In summary, while Threat Map currently stands as a functional prototype tailored for email threat visualization, its comparison with industry leaders reveals areas for significant future development. The core lessons learned from this analysis are the critical roles of automated, real-time data ingestion, the immense value of integrating external threat intelligence, the power of dynamic analysis (sandboxing), the potential for machine learning to automate classification and prioritize analyst workload, and the necessity of integrating response actions into the workflow. These insights provide concrete direction for evolving Threat Map into a more comprehensive and effective cybersecurity tool.

Testing and Quality Assurance

The development of the Threat Map application incorporated a strong emphasis on Testing and Quality Assurance (QA) to ensure the system was not only functional but also robust, secure, and delivered a reliable user experience. Given the project's scope and timeline as an internship, the QA strategy primarily relied on rigorous manual testing throughout the iterative development cycles, complemented by planning for future automated testing. This section details the QA approach, test coverage, bug resolution process, and lessons learned.

QA Strategy and Approach

The core QA strategy adopted was heavily influenced by an iterative, feature-by-feature development model. As each module (Dashboard, Reports, Admin Panel) and its associated functionalities (authentication, data fetching, CRUD operations, filtering, reporting) were built, extensive manual testing was performed. The approach can be characterized by the following:

- Exploratory Testing: Initially, developers and supervisors engaged in exploratory testing, interacting with new features freely to identify obvious bugs, usability issues, and unexpected behaviors. This was crucial in the early stages to quickly catch fundamental flaws and refine the user interface flow.
- Structured Manual Test Case Design: As features matured, a more structured approach was taken. Test cases were designed based on application requirements and expected user workflows. These cases detailed specific steps to perform, expected outcomes, and criteria for success. Documenting these cases allowed for repeatable testing and easier regression checks.
- Regression Testing: After significant bug fixes or the addition of new features, regression testing was performed manually. This involved re-running the documented test cases for existing functionalities to ensure that recent changes had not introduced new bugs or negatively impacted previously working parts of the application.
- Focus on User Workflows: Testing centered on the key user workflows identified for analysts and administrators. This ensured that core tasks, such as logging in, viewing threats, filtering the list, adding remarks, generating reports, and managing users, could be completed smoothly and correctly from end-to-end.

Test Coverage Areas

Testing activities spanned all major components and functionalities of the Threat Map application:

- Authentication and Authorization:
  - Logging in with valid/invalid credentials.
  - Accessing protected pages directly without authentication.
  - Session expiry handling (simulating token expiry or manual removal).
  - Attempting to access Admin Panel as a non-admin user.
  - Verifying conditional display of Admin link based on role.
- Data Fetching and Display:
  - Successful loading of data on Dashboard, Reports, and Admin pages.
  - Handling cases with no data ("No Records Found" messaging).
  - Correct transformation of raw backend data for UI display.
  - Verification of initial loading states (skeletons) and their removal.
  - Performance observation with moderate data volumes.
- UI Interactivity and Responsiveness:
  - Smooth transitions between different views (e.g., Dashboard list to detail).
  - Functionality of clickable elements (buttons, icons, table rows, column headers).
  - Correct behavior of modal dialogs (opening, closing, content display).
  - Basic responsiveness checks on different browser window sizes (though focused on desktop).
  - Keyboard navigation for critical elements and shortcuts (e.g., Escape key).
- Data Operations (CRUD - Create, Read, Update, Delete):
  - Adding, editing, and deleting users via Admin Panel (Admin-only access).
  - Adding and deleting allowed senders via Admin Panel.
  - Updating threat status (active/inactive) on the Dashboard detail view.
  - Adding and editing analyst remarks on the Dashboard detail view.
  - Verifying data persistence after successful operations (often by re-fetching and checking the UI).
- Filtering, Searching, and Sorting:
  - Applying single and multiple filters on Dashboard (by threat type).
  - Searching the Dashboard table by sender email (case-insensitive).
  - Sorting the Dashboard table by date (ascending/descending).
  - Filtering the Reports table by report type.

    - Verifying that filtering/searching/sorting correctly updates the displayed data subset.
- Report Generation and Download:
    - Opening and interacting with the Generate Report dialog.
    - Selecting different report parameters and verifying dynamic form updates.
    - Validating input parameters before report generation (e.g., date range logic).
    - Triggering report generation with valid criteria (expecting success message and table update).
    - Handling report generation attempts with no matching data.
    - Downloading generated reports via presigned URLs.
- Error Handling and Feedback:
    - Submitting forms with empty required fields (client-side validation).
    - Attempting actions when offline or backend is unavailable (simulated network errors).
    - Backend API errors (e.g., 400, 500 responses) displayed to the user.
    - Providing clear messages for successful operations and errors.
    - Preventing duplicate submissions (e.g., by disabling buttons).

Designing Reproducible Test Cases

To ensure consistency and enable regression testing, manual test cases were documented with enough detail to be followed by another tester. A typical test case structure included:

- Test Case ID: Unique identifier (e.g., TM-DASH-001)
- Feature/Module: Dashboard, Reports, Admin Panel, etc.
- Description: A brief summary of the functionality being tested (e.g., "Verify Dashboard loads and displays threat data").
- Preconditions: Any setup required before starting the test (e.g., "User is logged in as an Analyst", "Threat Map backend services are running and populated with data").
- Steps: A numbered list of specific actions to perform in the UI (e.g., "1. Navigate to the Dashboard page.", "2. Observe the main table area.", "3. Click on the 'Date' column header.").
- Expected Result: What should happen after performing the steps (e.g., "1. The page loads without errors and the 'LOADING' indicator appears.", "2. The table populates with threat data after a brief delay, and the 'LOADING' indicator

disappears.", "3. The table rows reorder by date, with the most recent entries at the top, and the arrow icon next to 'Date' shows a down arrow (↓).").
- Actual Result: What actually happened when the steps were performed.
- Status: Pass / Fail.
- Notes/Bugs: Any observations, deviations, or links to bug reports.

While this was a manual process, documenting these steps was invaluable for reproducing bugs and ensuring consistent verification of core features after code changes.

Manual vs. Automated Testing

Due to project constraints and the rapid pace of feature development, manual testing was the primary method employed. This allowed for immediate feedback on the UI/UX, workflow issues, and integration points as code was being written. Manual testing was particularly effective for:

- Validating the end-to-end user workflows across different modules.
- Assessing the visual presentation, animations, and responsiveness.
- Testing complex interactions involving multiple steps or conditional logic (like the report generation dialog).
- Exploratory testing of new or modified features.

However, manual testing is time-consuming and prone to human error, especially for regression testing large numbers of scenarios. It doesn't scale well as the application grows. The project identified that automated testing would be essential for long-term maintenance and faster, more reliable regression cycles. Automation was planned but not extensively implemented within the initial internship period.

Detailed Bug Log Analysis

Throughout the testing process, identified issues were logged, prioritized, and addressed. Analyzing these bugs provided critical insights into potential weaknesses in the codebase and development practices. Below are narratives for some notable bugs and their resolution, drawing from the original bug log summary:

Bug 1: Case-Sensitive Search on Dashboard

- Discovery: During manual testing of the Dashboard search functionality, it was observed that typing "sender@example.com" correctly filtered threats from that sender, but typing "Sender@example.com" or "sender@EXAMPLE.COM" yielded no results, even if a matching email existed in the data.
- Root Cause: The JavaScript function searchTable() was performing a direct substring match on the 'from' string using methods like `includes()` or `indexOf()` without normalizing case. The comparison `threat.from.includes(value)` failed if the case didn't match exactly.
- Resolution: The fix involved converting both the search input value and the 'from' attribute of each threat object to lowercase before performing the comparison. The code `threat.from.toLowerCase().includes(value.toLowerCase())` was implemented, ensuring the search became case-insensitive and matched users' natural typing behavior. This improved usability significantly.
- Testing Artifact (Pseudo-Code Fix):

```
// Old (Case-sensitive)
// return data.filter(threat => threat.from.includes(value));

// New (Case-insensitive)
return data.filter(threat =>
threat.from.toLowerCase().includes(value.toLowerCase()));
```

Bug 2: Multi-Line Remarks Not Preserving Line Breaks

- Discovery: When adding remarks with multiple lines in the threat detail editor, saving the remarks seemed to work, but when viewing the detail again or re-opening the editor, the line breaks were gone, and all text appeared on a single line.
- Root Cause: The application stored multi-line remarks as a single string in the database using a custom delimiter `&&&` to represent line breaks. However, the frontend logic for *displaying* remarks and *populating the editor* was not

correctly converting between actual newline characters (`\n`) and the `&&&` delimiter on both saving and loading. Initially, only loading handled the conversion, or the saving didn't correctly replace newlines entered in the textarea with `&&&`.

- Resolution: The fix involved ensuring a symmetrical conversion. When saving remarks, newline characters (`\n`) entered in the textarea were explicitly replaced with `&&&` before sending the string to the backend API. When populating the detail view and the edit textarea, the stored string was split by `&&&` and rejoined with newline characters (`\n`) for display. This two-way conversion preserved formatting.
- Testing Artifact (Pseudo-Code Fix Snippet):

```
// When saving remarks:
let remarksToSave = remarksInput.value.replace(/\n/g, '&&&');
// Send remarksToSave to backend

// When loading remarks for display/editor:
let remarksFromDB = threatData.remarks; // Assume fetched string from DB
let remarksForDisplay = remarksFromDB ? remarksFromDB.split('&&&').join('\n')
: '';
// Display remarksForDisplay or set textarea.value
```

Bug 3: Duplicate Report Generation on Rapid Clicks

- Discovery: When testing the "Generate Report" feature, rapid double-clicking on the "Confirm" button in the modal dialog could sometimes trigger the backend API call twice, potentially leading to duplicate report entries in the list.
- Root Cause: The button's click handler function was asynchronous (it involved an API call) but didn't immediately disable the button. If the user clicked again before the first API call completed and re-enabled the button, a second execution of the handler was triggered.
- Resolution: The fix was straightforward: immediately at the start of the asynchronous `generateReport` function, the "Confirm Generate" button element's `disabled` property was set to `true`. A `finally` block was added to the `try...catch` structure around the API call to ensure the button's `disabled` property was set back to `false` regardless of success or failure, making it ready for the next action. This prevented subsequent clicks while a request was in flight.

Bug 4: "No Records Found" Tooltip Persistence

- Discovery: In the Admin Panel, after deleting the last sender from the list, the "No Records Found" tooltip/message correctly appeared. However, if a new sender

was immediately added, the sender appeared in the table, but the "No Records Found" message sometimes lingered incorrectly.

- Root Cause: The logic for showing/hiding the "No Records Found" message relied on checking the number of rows in the table body after `bodyBuild()` completed. This check might not have been consistently triggered or correctly evaluated after the specific sequence of deleting the last item *then* adding a new one, which caused the table to transition from empty to non-empty state.
- Resolution: The fix involved explicitly calling the `isEmpty(tbody)` utility function (which toggles the visibility of the "No Records Found" element based on `tbody.rows.length`) consistently after *any* operation that modifies the table content, particularly after successful add/delete actions in the Admin panel. By ensuring `isEmpty()` was called immediately after re-fetching and rebuilding the Senders table, the message state was correctly synchronized with the actual table content.

Edge Case and Negative Testing

Beyond core functional flows, specific attention was paid to testing edge cases and negative scenarios to assess the application's robustness and error handling. Examples include:

- Invalid Input: Submitting forms with deliberately malformed data, such as using HTML tags

```
// Simplified report generation trigger after validation
async function generateReport(by, sender, threat_type, activity, fromDate,
toDate) {
    // Filter email_metadata based on parameters... (logic omitted)
    const required_Metadata = email_metadata.filter(/* ... filtering
logic ... */);

    if (required_Metadata.length === 0) {
        alert("No records found for the selected criteria.");
        return;
    }

    // Assuming backend generates PDF from this data
    const payload = required_Metadata;
    const endpoint = `generateReport?by=${by}`; // Pass type as query param
    const token = localStorage.getItem('id_token');

    try {
        // Disable button to prevent resubmission
        document.getElementById('confirmGen').disabled = true;
```

```
        const response = await fetch(endpoint, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${token}`
            },
            body: JSON.stringify(payload)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        alert("Report generated!!");
        closeDialog();
        // Re-fetch report list to show the new report
        fetchData('TABLE2').then(data => {
            reportData = data; // Update local data
            updateView(); // Refresh the UI table
        });

    } catch (error) {
        console.error('Report generation failed:', error);
        alert("Error generating report: " + error.message);
    } finally {
        // Re-enable button
        document.getElementById('confirmGen').disabled = false;
    }
}

// Simplified date sorting logic
dateHeader.addEventListener('click', function() {
    const currentOrder = this.getAttribute('data-order');
    const newOrder = currentOrder === 'desc' ? 'asc' : 'desc';
    this.setAttribute('data-order', newOrder);
    // Update arrow icon (assuming element with class 'sort-arrow')
    this.querySelector('.sort-arrow').textContent = newOrder === 'desc' ? '↓'
: '↑';

    // Sort the filtered data
    filteredData.sort((a, b) => {
        const dateA = new Date(a.date);
        const dateB = new Date(b.date);
        if (newOrder === 'desc') {
            return dateB - dateA; // Newest first
        } else {
            return dateA - dateB; // Oldest first
        }
    });

    // Rebuild the table with sorted data
```

```
    bodyBuild(threatsTableBody, filteredData);
});

// Simplified showLayout function
function showLayout(viewToShow, viewToHide) {
    // Add 'active' class to the view to show
    viewToShow.classList.add('active');
    // Remove 'active' class from the view to hide,
    // wait for transition to finish before setting display: none
    viewToHide.classList.remove('active');
    viewToHide.addEventListener('transitionend', function handler() {
        if (!viewToHide.classList.contains('active')) {
            viewToHide.style.display = 'none'; // Hide element after fade-out
        }
        viewToHide.removeEventListener('transitionend', handler);
    });
    // Assuming CSS handles the fade-in for .active
    viewToShow.style.display = 'flex'; // Or 'block', etc. based on layout
needs
}

// Simplified excerpt demonstrating action determination and payload
construction
async function adminAction(event) {
    const key = event.target.getAttribute('key');
    let payload = {};
    let endpoint = '';
    let method = 'PUT'; // Default method for updates/creates in this context

    if (key === 'create-user') {
        // Gather data from Create User form
        const username = document.getElementById('new-username').value;
        const email = document.getElementById('new-email').value;
        const tempPassword = document.getElementById('new-temp-
password').value;
        const role = document.getElementById('new-role').value;
        // Basic validation
        if (!username || !email || !tempPassword || !role) {
            alert("Please fill all required fields.");
            return;
        }
        payload = { username, email, tempPassword, group: role };
        endpoint = 'cognito-users-modification?key=create';
    } else if (key === 'update-user') {
        // Gather data from Edit User form
        const username = document.getElementById('edit-username').value; //
Assuming username is hidden but present
        const email = document.getElementById('edit-email').value;
        const status = document.getElementById('edit-status').value;
        if (!username || !email || !status) {
            alert("Please fill all required fields.");
```

```
            return;
        }
        payload = { username, email, status };
        endpoint = 'cognito-users-modification?key=modify';
    } // ... other actions (delete user, add/delete sender)

    // Send request to backend (simplified)
    try {
        const token = localStorage.getItem('id_token'); // Get JWT
        const response = await fetch(endpoint, {
            method: method,
            headers: {
                'Content-Type': 'application/json',
                'Authorization': `Bearer ${token}` // Attach JWT
            },
            body: JSON.stringify(payload)
        });
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }
        alert("Operation successful!");
        closeDialog(); // Close modal on success
        fetchData(); // Refresh UI data
    } catch (error) {
        console.error('Admin action failed:', error);
        alert("Operation failed: " + error.message);
    }
}
```

Conclusion

Enhanced Threat Monitoring: The Threat Map web application successfully enhances email-based threat monitoring in organisational settings.

Robust Security Architecture: Incorporates advanced encryption, real-time threat intelligence, and a scalable, modular design.

Operational Efficiency: User-friendly interfaces support swift incident response and actionable insights for cybersecurity teams.

Adaptability: Designed with scalability and adaptability to meet evolving cyber threat landscapes.

Future Work

Machine Learning Integration: Employ predictive threat analysis for improved anomaly detection and automated threat classification.

Example: Using ML algorithms to detect phishing patterns based on historical data.

Advanced Threat Intelligence Feeds: Integrate diverse external sources for enriched data analysis and enhanced detection accuracy.

Example: Connecting with global threat databases to identify known malicious IPs and domains.

Automated Response Playbooks: Develop automated workflows to reduce response times and minimise human errors.

Example: Automating email quarantine and user notifications when suspicious activities are detected.

Advanced Reporting Tools: Implement customisable dashboards and sophisticated data visualisation techniques.

Example: Interactive graphs depicting threat trends over time for management reports.